

OFFICIAL MICROSOFT LEARNING PRODUCT

10953A

HTML5 Programming

MCT USE ONLY. STUDENT USE PROHIBITED

Information in this document, including URL and other Internet Web site references, is subject to change without notice. Unless otherwise noted, the example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious, and no association with any real company, organization, product, domain name, e-mail address, logo, person, place or event is intended or should be inferred. Complying with all applicable copyright laws is the responsibility of the user. Without limiting the rights under copyright, no part of this document may be reproduced, stored in or introduced into a retrieval system, or transmitted in any form or by any means (electronic, mechanical, photocopying, recording, or otherwise), or for any purpose, without the express written permission of Microsoft Corporation.

Microsoft may have patents, patent applications, trademarks, copyrights, or other intellectual property rights covering subject matter in this document. Except as expressly provided in any written license agreement from Microsoft, the furnishing of this document does not give you any license to these patents, trademarks, copyrights, or other intellectual property.

The names of manufacturers, products, or URLs are provided for informational purposes only and Microsoft makes no representations and warranties, either expressed, implied, or statutory, regarding these manufacturers or the use of the products with any Microsoft technologies. The inclusion of a manufacturer or product does not imply endorsement of Microsoft of the manufacturer or product. Links may be provided to third party sites. Such sites are not under the control of Microsoft and Microsoft is not responsible for the contents of any linked site or any link contained in a linked site, or any changes or updates to such sites. Microsoft is not responsible for webcasting or any other form of transmission received from any linked site. Microsoft is providing these links to you only as a convenience, and the inclusion of any link does not imply endorsement of Microsoft of the site or the products contained therein.

© 2012 Microsoft Corporation. All rights reserved.

Microsoft and the trademarks listed at

<http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx> are trademarks of the Microsoft group of companies. All other trademarks are property of their respective owners.

Product Number: 10953A

Part Number: X18-22906

Released: 04/2012

MICROSOFT LICENSE TERMS
OFFICIAL MICROSOFT LEARNING PRODUCTS
MICROSOFT OFFICIAL COURSE Pre-Release and Final Release Versions

These license terms are an agreement between Microsoft Corporation and you. Please read them. They apply to the Licensed Content named above, which includes the media on which you received it, if any. These license terms also apply to any updates, supplements, internet based services and support services for the Licensed Content, unless other terms accompany those items. If so, those terms apply.

BY DOWNLOADING OR USING THE LICENSED CONTENT, YOU ACCEPT THESE TERMS. IF YOU DO NOT ACCEPT THEM, DO NOT DOWNLOAD OR USE THE LICENSED CONTENT.

If you comply with these license terms, you have the rights below.

1. DEFINITIONS.

- a. "Authorized Learning Center" means a Microsoft Learning Competency Member, Microsoft IT Academy Program Member, or such other entity as Microsoft may designate from time to time.
- b. "Authorized Training Session" means the Microsoft-authorized instructor-led training class using only MOC Courses that are conducted by a MCT at or through an Authorized Learning Center.
- c. "Classroom Device" means one (1) dedicated, secure computer that you own or control that meets or exceeds the hardware level specified for the particular MOC Course located at your training facilities or primary business location.
- d. "End User" means an individual who is (i) duly enrolled for an Authorized Training Session or Private Training Session, (ii) an employee of a MPN Member, or (iii) a Microsoft full-time employee.
- e. "Licensed Content" means the MOC Course and any other content accompanying this agreement. Licensed Content may include (i) Trainer Content, (ii) software, and (iii) associated media.
- f. "Microsoft Certified Trainer" or "MCT" means an individual who is (i) engaged to teach a training session to End Users on behalf of an Authorized Learning Center or MPN Member, (ii) currently certified as a Microsoft Certified Trainer under the Microsoft Certification Program, and (iii) holds a Microsoft Certification in the technology that is the subject of the training session.
- g. "Microsoft IT Academy Member" means a current, active member of the Microsoft IT Academy Program.
- h. "Microsoft Learning Competency Member" means a Microsoft Partner Network Program Member in good standing that currently holds the Learning Competency status.
- i. "Microsoft Official Course" or "MOC Course" means the Official Microsoft Learning Product instructor-led courseware that educates IT professionals or developers on Microsoft technologies.

- j. “Microsoft Partner Network Member” or “MPN Member” means a silver or gold-level Microsoft Partner Network program member in good standing.
 - k. “Personal Device” means one (1) device, workstation or other digital electronic device that you personally own or control that meets or exceeds the hardware level specified for the particular MOC Course.
 - l. “Private Training Session” means the instructor-led training classes provided by MPN Members for corporate customers to teach a predefined learning objective. These classes are not advertised or promoted to the general public and class attendance is restricted to individuals employed by or contracted by the corporate customer.
 - m. “Trainer Content” means the trainer version of the MOC Course and additional content designated solely for trainers to use to teach a training session using a MOC Course. Trainer Content may include Microsoft PowerPoint presentations, instructor notes, lab setup guide, demonstration guides, beta feedback form and trainer preparation guide for the MOC Course. To clarify, Trainer Content does not include virtual hard disks or virtual machines.
2. **INSTALLATION AND USE RIGHTS.** The Licensed Content is licensed not sold. The Licensed Content is licensed on a one copy per user basis, such that you must acquire a license for each individual that accesses or uses the Licensed Content.
- 2.1 Below are four separate sets of installation and use rights. Only one set of rights apply to you.
- a. **If you are a Authorized Learning Center:**
 - i. If the Licensed Content is in digital format for each license you acquire you may either:
 - 1. install one (1) copy of the Licensed Content in the form provided to you on a dedicated, secure server located on your premises where the Authorized Training Session is held for access and use by one (1) End User attending the Authorized Training Session, or by one (1) MCT teaching the Authorized Training Session, **or**
 - 2. install one (1) copy of the Licensed Content in the form provided to you on one (1) Classroom Device for access and use by one (1) End User attending the Authorized Training Session, or by one (1) MCT teaching the Authorized Training Session.
 - ii. You agree that:
 - 1. you will acquire a license for each End User and MCT that accesses the Licensed Content,
 - 2. each End User and MCT will be presented with a copy of this agreement and each individual will agree that their use of the Licensed Content will be subject to these license terms prior to their accessing the Licensed Content. Each individual will be required to denote their acceptance of the EULA in a manner that is enforceable under local law prior to their accessing the Licensed Content,
 - 3. for all Authorized Training Sessions, you will only use qualified MCTs who hold the applicable competency to teach the particular MOC Course that is the subject of the training session,
 - 4. you will not alter or remove any copyright or other protective notices contained in the Licensed Content,

5. you will remove and irretrievably delete all Licensed Content from all Classroom Devices and servers at the end of the Authorized Training Session,
6. you will only provide access to the Licensed Content to End Users and MCTs,
7. you will only provide access to the Trainer Content to MCTs, and
8. any Licensed Content installed for use during a training session will be done in accordance with the applicable classroom set-up guide.

b. If you are a MPN Member.

- i. If the Licensed Content is in digital format for each license you acquire you may either:
 1. install one (1) copy of the Licensed Content in the form provided to you on (A) one (1) Classroom Device, or (B) one (1) dedicated, secure server located at your premises where the training session is held for use by one (1) of your employees attending a training session provided by you, or by one (1) MCT that is teaching the training session, **or**
 2. install one (1) copy of the Licensed Content in the form provided to you on one (1) Classroom Device for use by one (1) End User attending a Private Training Session, or one (1) MCT that is teaching the Private Training Session.
- ii. You agree that:
 1. you will acquire a license for each End User and MCT that accesses the Licensed Content,
 2. each End User and MCT will be presented with a copy of this agreement and each individual will agree that their use of the Licensed Content will be subject to these license terms prior to their accessing the Licensed Content. Each individual will be required to denote their acceptance of the EULA in a manner that is enforceable under local law prior to their accessing the Licensed Content,
 3. for all training sessions, you will only use qualified MCTs who hold the applicable competency to teach the particular MOC Course that is the subject of the training session,
 4. you will not alter or remove any copyright or other protective notices contained in the Licensed Content,
 5. you will remove and irretrievably delete all Licensed Content from all Classroom Devices and servers at the end of each training session,
 6. you will only provide access to the Licensed Content to End Users and MCTs,
 7. you will only provide access to the Trainer Content to MCTs, and
 8. any Licensed Content installed for use during a training session will be done in accordance with the applicable classroom set-up guide.

c. If you are an End User:

You may use the Licensed Content solely for your personal training use. If the Licensed Content is in digital format, for each license you acquire you may (i) install one (1) copy of the Licensed Content in the form provided to you on one (1) Personal Device and install another copy on another Personal Device as a backup copy, which may be used only to reinstall the Licensed Content; or (ii) print one (1) copy of the Licensed Content. You may not install or use a copy of the Licensed Content on a device you do not own or control.

d. **If you are a MCT.**

- i. For each license you acquire, you may use the Licensed Content solely to prepare and deliver an Authorized Training Session or Private Training Session. For each license you acquire, you may install and use one (1) copy of the Licensed Content in the form provided to you on one (1) Personal Device and install one (1) additional copy on another Personal Device as a backup copy, which may be used only to reinstall the Licensed Content. You may not install or use a copy of the Licensed Content on a device you do not own or control.
- ii. **Use of Instructional Components in Trainer Content.** You may customize, in accordance with the most recent version of the MCT Agreement, those portions of the Trainer Content that are logically associated with instruction of a training session. If you elect to exercise the foregoing rights, you agree: (a) that any of these customizations will only be used for providing a training session, (b) any customizations will comply with the terms and conditions for Modified Training Sessions and Supplemental Materials in the most recent version of the MCT agreement and with this agreement. For clarity, any use of “customize” refers only to changing the order of slides and content, and/or not using all the slides or content, it does not mean changing or modifying any slide or content.

2.2 Separation of Components. The Licensed Content components are licensed as a single unit and you may not separate the components and install them on different devices.

2.3 Reproduction/Redistribution Licensed Content. Except as expressly provided in the applicable installation and use rights above, you may not reproduce or distribute the Licensed Content or any portion thereof (including any permitted modifications) to any third parties without the express written permission of Microsoft.

2.4 Third Party Programs. The Licensed Content may contain third party programs or services. These license terms will apply to your use of those third party programs or services, unless other terms accompany those programs and services.

2.5 Additional Terms. Some Licensed Content may contain components with additional terms, conditions, and licenses regarding its use. Any non-conflicting terms in those conditions and licenses also apply to that respective component and supplements the terms described in this Agreement.

3. **PRE-RELEASE VERSIONS.** If the Licensed Content is a pre-release (“beta”) version, in addition to the other provisions in this agreement, then these terms also apply:

- a. **Pre-Release Licensed Content.** This Licensed Content is a pre-release version. It may not contain the same information and/or work the way a final version of the Licensed Content will. We may change it for the final version. We also may not release a final version. Microsoft is under no obligation to provide you with any further content, including the final release version of the Licensed Content.
- b. **Feedback.** If you agree to give feedback about the Licensed Content to Microsoft, either directly or through its third party designee, you give to Microsoft without charge, the right to use, share and commercialize your feedback in any way and for any purpose. You also give to third parties, without charge, any patent rights needed for their products, technologies and services to use or interface with any specific parts of a Microsoft software, Microsoft product, or service that includes the feedback. You will not give feedback that is subject to a license that requires Microsoft to license its software, technologies, or products to third parties because we include your feedback in them. These rights

survive this agreement.

- c. **Term.** If you are an Authorized Training Center, MCT or MPN, you agree to cease using all copies of the beta version of the Licensed Content upon (i) the date which Microsoft informs you is the end date for using the beta version, or (ii) sixty (60) days after the commercial release of the Licensed Content, whichever is earliest ("**beta term**"). Upon expiration or termination of the beta term, you will irretrievably delete and destroy all copies of same in the possession or under your control.
4. **INTERNET-BASED SERVICES.** Microsoft may provide Internet-based services with the Licensed Content, which may change or be canceled at any time.
 - a. **Consent for Internet-Based Services.** The Licensed Content may connect to computer systems over an Internet-based wireless network. In some cases, you will not receive a separate notice when they connect. Using the Licensed Content operates as your consent to the transmission of standard device information (including but not limited to technical information about your device, system and application software, and peripherals) for internet-based services.
 - b. **Misuse of Internet-based Services.** You may not use any Internet-based service in any way that could harm it or impair anyone else's use of it. You may not use the service to try to gain unauthorized access to any service, data, account or network by any means.
5. **SCOPE OF LICENSE.** The Licensed Content is licensed, not sold. This agreement only gives you some rights to use the Licensed Content. Microsoft reserves all other rights. Unless applicable law gives you more rights despite this limitation, you may use the Licensed Content only as expressly permitted in this agreement. In doing so, you must comply with any technical limitations in the Licensed Content that only allows you to use it in certain ways. Except as expressly permitted in this agreement, you may not:
 - install more copies of the Licensed Content on devices than the number of licenses you acquired;
 - allow more individuals to access the Licensed Content than the number of licenses you acquired;
 - publicly display, or make the Licensed Content available for others to access or use;
 - install, sell, publish, transmit, encumber, pledge, lend, copy, adapt, link to, post, rent, lease or lend, make available or distribute the Licensed Content to any third party, except as expressly permitted by this Agreement.
 - reverse engineer, decompile, remove or otherwise thwart any protections or disassemble the Licensed Content except and only to the extent that applicable law expressly permits, despite this limitation;
 - access or use any Licensed Content for which you are not providing a training session to End Users using the Licensed Content;
 - access or use any Licensed Content that you have not been authorized by Microsoft to access and use; or
 - transfer the Licensed Content, in whole or in part, or assign this agreement to any third party.
6. **RESERVATION OF RIGHTS AND OWNERSHIP.** Microsoft reserves all rights not expressly granted to you in this agreement. The Licensed Content is protected by copyright and other intellectual property laws and treaties. Microsoft or its suppliers own the title, copyright, and other intellectual property rights in the Licensed Content. You may not remove or obscure any copyright, trademark or patent notices that appear on the Licensed Content or any components thereof, as delivered to you.

7. **EXPORT RESTRICTIONS.** The Licensed Content is subject to United States export laws and regulations. You must comply with all domestic and international export laws and regulations that apply to the Licensed Content. These laws include restrictions on destinations, End Users and end use. For additional information, see www.microsoft.com/exporting.
8. **LIMITATIONS ON SALE, RENTAL, ETC. AND CERTAIN ASSIGNMENTS.** You may not sell, rent, lease, lend or sublicense the Licensed Content or any portion thereof, or transfer or assign this agreement.
9. **SUPPORT SERVICES.** Because the Licensed Content is “as is”, we may not provide support services for it.
10. **TERMINATION.** Without prejudice to any other rights, Microsoft may terminate this agreement if you fail to comply with the terms and conditions of this agreement. Upon any termination of this agreement, you agree to immediately stop all use of and to irretrievable delete and destroy all copies of the Licensed Content in your possession or under your control.
11. **LINKS TO THIRD PARTY SITES.** You may link to third party sites through the use of the Licensed Content. The third party sites are not under the control of Microsoft, and Microsoft is not responsible for the contents of any third party sites, any links contained in third party sites, or any changes or updates to third party sites. Microsoft is not responsible for webcasting or any other form of transmission received from any third party sites. Microsoft is providing these links to third party sites to you only as a convenience, and the inclusion of any link does not imply an endorsement by Microsoft of the third party site.
12. **ENTIRE AGREEMENT.** This agreement, and the terms for supplements, updates and support services are the entire agreement for the Licensed Content.
13. **APPLICABLE LAW.**
 - a. United States. If you acquired the Licensed Content in the United States, Washington state law governs the interpretation of this agreement and applies to claims for breach of it, regardless of conflict of laws principles. The laws of the state where you live govern all other claims, including claims under state consumer protection laws, unfair competition laws, and in tort.
 - b. Outside the United States. If you acquired the Licensed Content in any other country, the laws of that country apply.
14. **LEGAL EFFECT.** This agreement describes certain legal rights. You may have other rights under the laws of your country. You may also have rights with respect to the party from whom you acquired the Licensed Content. This agreement does not change your rights under the laws of your country if the laws of your country do not permit it to do so.
15. **DISCLAIMER OF WARRANTY. THE LICENSED CONTENT IS LICENSED "AS-IS," "WITH ALL FAULTS," AND "AS AVAILABLE." YOU BEAR THE RISK OF USING IT. MICROSOFT CORPORATION AND ITS RESPECTIVE AFFILIATES GIVE NO EXPRESS WARRANTIES, GUARANTEES, OR CONDITIONS UNDER OR IN RELATION TO THE LICENSED CONTENT. YOU MAY HAVE ADDITIONAL CONSUMER RIGHTS UNDER YOUR LOCAL LAWS WHICH THIS AGREEMENT CANNOT CHANGE. TO THE EXTENT PERMITTED UNDER YOUR LOCAL LAWS, MICROSOFT CORPORATION AND ITS RESPECTIVE AFFILIATES EXCLUDE ANY IMPLIED WARRANTIES OR CONDITIONS, INCLUDING THOSE OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT.**

16. **LIMITATION ON AND EXCLUSION OF REMEDIES AND DAMAGES. TO THE EXTENT NOT PROHIBITED BY LAW, YOU CAN RECOVER FROM MICROSOFT CORPORATION AND ITS SUPPLIERS ONLY DIRECT DAMAGES UP TO USD\$5.00. YOU AGREE NOT TO SEEK TO RECOVER ANY OTHER DAMAGES, INCLUDING CONSEQUENTIAL, LOST PROFITS, SPECIAL, INDIRECT OR INCIDENTAL DAMAGES FROM MICROSOFT CORPORATION AND ITS RESPECTIVE SUPPLIERS.**

This limitation applies to

- anything related to the Licensed Content, services made available through the Licensed Content, or content (including code) on third party Internet sites or third-party programs; and
- claims for breach of contract, breach of warranty, guarantee or condition, strict liability, negligence, or other tort to the extent permitted by applicable law.

It also applies even if Microsoft knew or should have known about the possibility of the damages. The above limitation or exclusion may not apply to you because your country may not allow the exclusion or limitation of incidental, consequential or other damages.

Please note: As this Licensed Content is distributed in Quebec, Canada, some of the clauses in this agreement are provided below in French.

Remarque : Ce le contenu sous licence étant distribué au Québec, Canada, certaines des clauses dans ce contrat sont fournies ci-dessous en français.

EXONÉRATION DE GARANTIE. Le contenu sous licence visé par une licence est offert « tel quel ». Toute utilisation de ce contenu sous licence est à votre seule risque et péril. Microsoft n'accorde aucune autre garantie expresse. Vous pouvez bénéficier de droits additionnels en vertu du droit local sur la protection des consommateurs, que ce contrat ne peut modifier. La ou elles sont permises par le droit locale, les garanties implicites de qualité marchande, d'adéquation à un usage particulier et d'absence de contrefaçon sont exclues.

LIMITATION DES DOMMAGES-INTÉRÊTS ET EXCLUSION DE RESPONSABILITÉ POUR LES DOMMAGES. Vous pouvez obtenir de Microsoft et de ses fournisseurs une indemnisation en cas de dommages directs uniquement à hauteur de 5,00 \$ US. Vous ne pouvez prétendre à aucune indemnisation pour les autres dommages, y compris les dommages spéciaux, indirects ou accessoires et pertes de bénéfices.

Cette limitation concerne:

- tout ce qui est relié au le contenu sous licence , aux services ou au contenu (y compris le code) figurant sur des sites Internet tiers ou dans des programmes tiers ; et
- les réclamations au titre de violation de contrat ou de garantie, ou au titre de responsabilité stricte, de négligence ou d'une autre faute dans la limite autorisée par la loi en vigueur.

Elle s'applique également, même si Microsoft connaissait ou devrait connaître l'éventualité d'un tel dommage. Si votre pays n'autorise pas l'exclusion ou la limitation de responsabilité pour les dommages indirects, accessoires ou de quelque nature que ce soit, il se peut que la limitation ou l'exclusion ci-dessus ne s'appliquera pas à votre égard.

EFFET JURIDIQUE. Le présent contrat décrit certains droits juridiques. Vous pourriez avoir d'autres droits prévus par les lois de votre pays. Le présent contrat ne modifie pas les droits que vous confèrent les lois de votre pays si celles-ci ne le permettent pas.

Revised December 2011

Welcome!

Thank you for taking our training! We've worked together with our Microsoft Certified Partners for Learning Solutions and our Microsoft IT Academies to bring you a world-class learning experience—whether you're a professional looking to advance your skills or a student preparing for a career in IT.

- **Microsoft Certified Trainers and Instructors**—Your instructor is a technical and instructional expert who meets ongoing certification requirements. And, if instructors are delivering training at one of our Certified Partners for Learning Solutions, they are also evaluated throughout the year by students and by Microsoft.
- **Certification Exam Benefits**—After training, consider taking a Microsoft Certification exam. Microsoft Certifications validate your skills on Microsoft technologies and can help differentiate you when finding a job or boosting your career. In fact, independent research by IDC concluded that 75% of managers believe certifications are important to team performance¹. Ask your instructor about Microsoft Certification exam promotions and discounts that may be available to you.
- **Customer Satisfaction Guarantee**—Our Certified Partners for Learning Solutions offer a satisfaction guarantee and we hold them accountable for it. At the end of class, please complete an evaluation of today's experience. We value your feedback!

We wish you a great learning experience and ongoing success in your career!

Sincerely,

Microsoft Learning
www.microsoft.com/learning

Microsoft | Learning

¹ IDC, Value of Certification: Team Certification and Organizational Performance, November 2006

Acknowledgments

Microsoft Learning would like to acknowledge and thank the following for their contribution towards developing this title. Their effort at various stages in the development has ensured that you have a good classroom experience.

Ishai Ram – Lead Content Developer

Ishai is the Vice President of SELA Group. He has over 20 years of experience as a professional trainer and consultant on computer software and electronics.

Gil Fink - Subject Matter Expert

Gil is a senior architect at Sela Group, and a Microsoft MVP. He is an expert in Web development and data platform. He is consulting for enterprises and companies, where he architects and develops Web and RIA-based solutions.

Alex Golesh - Subject Matter Expert

Alex is a Senior Architect at Sela Group and a Microsoft MVP. He is an international expert of Silverlight, WPF, and Windows Phone. He has been developing training samples and courses for various product groups in Microsoft Corp.

Noam Kfir - Subject Matter Expert

Noam is a Senior Consultant at Sela Group. With over a decade of experience in developing Web applications, he is one of Sela's senior consultants on HTML5 and other Web technologies.

Sela Team – Content Contributors

Baruch Toledano, Roman Podobryanski, and Daniel Sterman

Toi B. Wright – Technical Reviewer

Toi is an independent consultant who has been working as a software developer for more than 25 years. She is a Microsoft MVP in ASP/ASP.NET and the Founder and President of the Dallas ASP.NET User Group.

Contents

Module 1: Introduction to HTML5 Development

Module Overview	1-2
Lesson 1: Overview of HTML	1-3
Lesson 2: What's New in HTML5?	1-26
Lesson 3: Introduction to CSS	1-31
Lesson 4: Introduction to JavaScript	1-55
Lesson 5: Introduction to jQuery	1-98
Lab: Building a Basic Webpage in the Visual Studio 2010 SP1 Environment	1-117
Module Review and Takeaways	1-124

Module 2: Creating Page Structure and Navigation

Module Overview	2-2
Lesson 1: HTML5 Structural Elements	2-3
Lesson 2: Navigation and Menus	2-16
Lab: Creating Page Structure and Navigation	2-30
Module Review and Takeaways	2-33

Module 3: Creating Form Input and Validation

Module Overview	3-2
Lesson 1: Working with Input Types	3-3
Lesson 2: Using Form Attributes	3-16
Lesson 3: Validation	3-35
Lesson 4: Using Browser Detection, Feature Detection, and Modernizr	3-44
Lab: Creating Form Input and Validation	3-55
Module Review and Takeaways	3-59

Module 4: Laying Out and Styling Webpages

Module Overview	4-2
Lesson 1: Creating Layouts	4-3
Lesson 2: Advanced CSS by Using CSS3	4-17
Lab: Laying Out and Styling Webpages	4-35
Module Review and Takeaways	4-37

Module 5: Getting Started with Graphics and Multimedia Elements

Module Overview	5-2
Lesson 1: Canvas Basics	5-4
Lesson 2: Video/Audio Formats and Codecs	5-51
Lesson 3: Controlling Multimedia with JavaScript	5-71
Lab: Getting Started with Graphics and Multimedia Elements	5-86

Module Review and Takeaways	5-89
Module 6: Creating Advanced Graphics	
Module Overview	6-2
Lesson 1: Drawing with SVG	6-3
Lesson 2: Animation	6-22
Lesson 3: Choosing Between SVG and Canvas	6-39
Lab: Creating Advanced Graphics	6-43
Module Review and Takeaways	6-46
Module 7: Using Client-Side Storage	
Module Overview	7-2
Lesson 1: Web Storage vs. Cookies	7-3
Lesson 2: Web Storage API	7-9
Lab: Using Client-Side Storage	7-18
Module Review and Takeaways	7-21
Module 8: Using Advanced HTML5 JavaScript APIs	
Module Overview	8-2
Lesson 1: Using the Drag-and-Drop API	8-3
Lesson 2: File API	8-14
Lesson 3: Geolocation API	8-28
Lab: Using Advanced HTML5 JavaScript APIs	8-38
Module Review and Takeaways	8-42
Module 9: Using WebMatrix and Other Developer Tools	
Module Overview	9-2
Lesson 1: WebMatrix	9-4
Lesson 2: ASP.NET MVC 3 and Razor	9-9
Lab: Using WebMatrix and Other Development Tools	9-18
Module Review and Takeaways	9-21

Lab Answer Keys

About This Course

This section provides you with a brief description of the course, audience, suggested prerequisites, and course objectives.

Course Description

This three-day instructor-led course provides participants with the knowledge and skills to develop HTML5 web applications using Microsoft Visual Studio® 2010.

Audience

The target audience for this training course is entry-level developers for HTML who have a basic but limited understanding of HTML and who are interested in learning core HTML5, CSS, and JavaScript programming skills that are required to successfully create applications for the upcoming Internet Explorer 10 browser and the next version of the Windows operating system.

Student Prerequisites

This course requires that you meet the following prerequisites:

- You are expected to have three to six months of development experience in a professional environment.
- You are expected to be familiar with the HTML document structure and associated terminology, but are not expected to be expert and do not need to have experience with HTML5.
- One to three months' experience with HTML and XML, as well as a general familiarity with the Visual Studio IDE, are recommended.

Course Objectives

After completing this course, students will be able to:

- Create and view a simple Web application using HTML within the Visual Studio 2010 SP1 IDE.
- Explain the new semantic and structural elements that can be used to create HTML5 Webpages.
- Create Web forms that make use of validation capabilities using the new input types and attributes included in the HTML5 specifications.
- Create layouts and styles using advanced CSS and CSS3.
- Integrate graphics and multimedia into webpages using the Canvas, SVG, Video and Audio elements.
- Implement offline data scenarios using the new HTML5 Web Storage API.
- Use advanced HTML5 JavaScript APIs such as drag-and-drop, File API and Geolocation API.
- Create simple web applications using WebMatrix or ASP.NET MVC.

Course Outline

This section provides an outline of the course:

Module 1, "Introduction to HTML5 Development" explains the students how to create a simple HTML web application using Visual Studio 2010. The students will get familiar with HTML, JavaScript, CSS and jQuery library.

Module 2, "Creating Page Structure and Navigation" describes the new HTML5 structural elements. It also explain webpages navigation concepts and HTML5 navigation elements.

Module 3, "Creating Form Input and Validation" explains what are forms and the slew of new HTML5 input controls and capabilities. It is also introduces the new HTML5 attributes. At the end there is a discussion about feature detection which helps to detect whether a HTML5 feature is available to use.

Module 4, "Laying Out and Styling Webpages" teaches Webpage layout concepts and describe CSS3, which reflects a significant change from previous versions of HTML and CSS.

Module 5, "Getting Started with Graphics and Multimedia Elements" explain the basics of working with graphics and multimedia elements such as the canvas, video and audio. Students learn how to draw basic shapes before they learn how to place images and video on the canvas.

Module 6, "Creating Advanced Graphics" describes SVG and some of its concepts to the students. The module also compares the canvas element with SVG and explains how to create animation with both of the elements.

Module 7, "Using Client-Side Storage" explain the students about the new HTML5 Web Storage API, which solves a great many common problems today and enables storing data on the client-side.

Module 8, "Using Advanced HTML5 JavaScript APIs" demonstrates a few key HTML5 JavaScript APIs, including the Drag-and-Drop API, the File API and the Geolocation API. The module explains the APIs briefly.

Module 9, "Using WebMatrix and Other Developer Tools" Introduce students to the Microsoft Web Platform, WebMatrix and its abilities, ASP.NET MVC 3 and the Razor view engine as the main frameworks for building Web applications.

Course Materials

The following materials are included with your kit:

- **Course Handbook** A succinct classroom learning guide that provides all the critical technical information in a crisp, tightly-focused format, which is just right for an effective in-class learning experience.
 - **Lessons:** Guide you through the learning objectives and provide the key points that are critical to the success of the in-class learning experience.
 - **Labs:** Provide a real-world, hands-on platform for you to apply the knowledge and skills learned in the module.
 - **Module Reviews and Takeaways:** Provide improved on-the-job reference material to boost knowledge and skills retention.
 - **Lab Answer Keys:** Provide step-by-step lab solution guidance at your fingertips when it's needed.



Course Companion Content on the <http://www.microsoft.com/learning/companionmoc/> Site: Searchable, easy-to-navigate digital content with integrated premium on-line resources designed to supplement the Course Handbook.

- **Modules:** Include companion content, such as questions and answers, detailed demo steps and additional reading links, for each lesson. Additionally, they include Lab Review questions and answers and Module Reviews and Takeaways sections, which contain the review questions and answers, best practices, common issues and troubleshooting tips with answers, and real-world issues and scenarios with answers.
- **Resources:** Include well-categorized additional resources that give you immediate access to the most up-to-date premium content on TechNet, MSDN®, Microsoft Press®.



Student Course files on the <http://www.microsoft.com/learning/companionmoc/> Site: Includes the Allfiles.exe, a self-extracting executable file that contains all the files required for the labs and demonstrations.

- **Course evaluation** At the end of the course, you will have the opportunity to complete an online evaluation to provide feedback on the course, training facility, and instructor.
 - To provide additional comments or feedback on the course, send e-mail to support@mscourseware.com. To inquire about the Microsoft Certification Program, send e-mail to mcphelp@microsoft.com.

Virtual Machine Environment

This section provides the information for setting up the classroom environment to support the business scenario of the course.

Virtual Machine Configuration

In this course, you will use Hyper-V™ to perform the labs. Each classroom computer serves as a host for one virtual machine that will run in Hyper-V. There is one Hyper-V virtual machine for all the labs in this course, named 10953A-SEA-DEV.

Important At the end of each lab, you must close the virtual machine and must not save any changes. To close a virtual machine without saving the changes, perform the following steps: 1. On the virtual machine, on the Action menu, click Close. 2. In the Close dialog box, in the What do you want the virtual machine to do? list, click Turn off and delete changes, and then click OK.

The following table shows the role of each virtual machine used in this course:

Virtual machine	Role
Base10C-W7-HV.vhd	Base
MT11-SEA-DEV10.vhd	Middle Tier
UT11-SEA-DEV10.vhd	Upper Tier
10953A-Allfiles.vhd	Lab Files

Software Configuration

The following software is installed on each VM:

- Windows 7
- Microsoft® Visual Studio® 2010
- Microsoft SQL Server® 2008
- Internet Explorer 9
- IIS 7.5 Express
- WebMatrix
- Internet Explorer 10
- Entity Framework 4.1
- Web Platform installer

Course Files

There are files associated with the labs in this course. The lab files are located in the folder D:\Mod0X\Labfiles\ on the student computers.

Classroom Setup

Each classroom computer will have the same virtual machine configured in the same way.

Course Hardware Level

To ensure a satisfactory student experience, Microsoft Learning requires a minimum equipment configuration for trainer and student computers in all Microsoft Certified Partner for Learning Solutions (CPLS) classrooms in which Official Microsoft Learning Product courseware are taught.

The classroom computers require the following hardware configuration:

- Hardware Level 6
- Intel Virtualization Technology (Intel VT) or AMD Virtualization (AMD-V) processor
- Dual 120 GB hard disks 7200 RM SATA or better*

- 4 GB RAM expandable to 8GB or higher
- DVD drive
- Network adapter
- Super VGA (SVGA) 17-inch monitor
- Microsoft Mouse or compatible pointing device
- Sound card with amplified speakers

*Striped

In addition, the instructor computer must be connected to a projection display device that supports SVGA 1024 x 768 pixels, 16 bit colors.

MCT USE ONLY. STUDENT USE PROHIBITED

Module 1

Introduction to HTML5 Development

Contents:

Module Overview	1-2
Lesson 1: Overview of HTML	1-3
Lesson 2: What's New in HTML5?	1-26
Lesson 3: Introduction to CSS	1-31
Lesson 4: Introduction to JavaScript	1-55
Lesson 5: Introduction to jQuery	1-98
Lab: Building a Basic Webpage in the Visual Studio 2010 SP1 Environment	1-117
Module Review and Takeaways	1-121

Module Overview

- Overview of HTML
- What's New in HTML5?
- Introduction to CSS
- Introduction to JavaScript
- Introduction to jQuery

HTML is the markup language that every web developer uses to structure and present content on the Internet. Cascading Style Sheets (CSS) is used to add styling and layout to webpages built by using HTML. JavaScript is the scripting language that takes static HTML and makes it dynamic.

This module introduces HTML development, and presents all the basics needed to build a website or application. It also introduces CSS and JavaScript, which with HTML are the basis for web development. In addition, the module introduces the jQuery JavaScript library, the most popular JavaScript library at the time of writing.

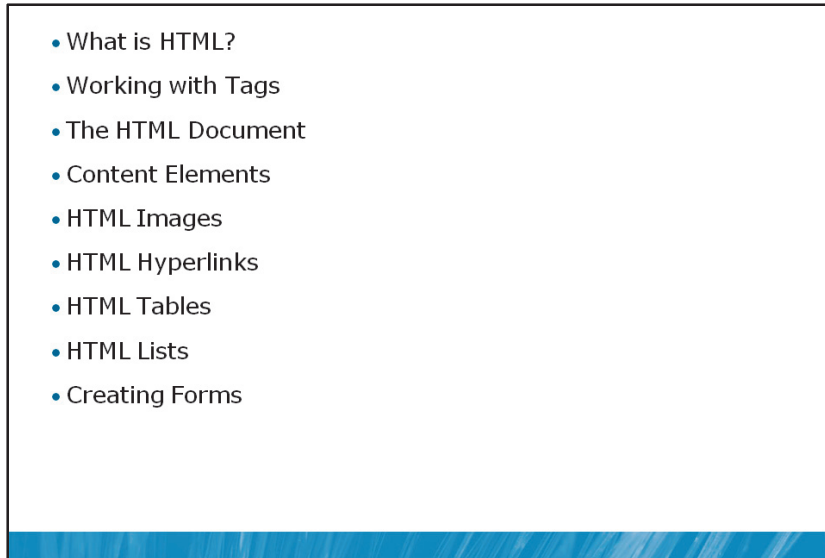
Objectives

After completing this module, you will be able to:

- Explain HTML basics.
- Explain what HTML5 is
- Explain CSS basics.
- Explain JavaScript basics.
- Explain the jQuery Library.
- Identify the essential features of Visual Studio 2010 SP1 that are required to start developing HTML5 web application projects.

Lesson 1

Overview of HTML



HTML is the building block of all the websites and applications that exists on the Internet. It is used, since its inception in 1991, to create webpage structures. It is very simple and easy to learn and can be used to create sophisticated website similar to Facebook or Twitter.

This lesson is dedicated to basic HTML development. It contains information about what is HTML, what are HTML tags/elements and what are the main elements in HTML. Mastering HTML can help you to understand how to design the layout of webpages and create better websites and applications.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain core HTML basics.
- Explain HTML tags.
- Describe HTML document.
- Describe HTML main elements.
- Describe HTML images.
- Describe HTML hyperlinks.
- Describe HTML tables.
- Describe HTML lists.
- Explain how to create HTML forms.

What is HTML?

- Invented in 1991 by Tim Berners-Lee
- HTML = Hyper Text Markup Language
- Contains data and metadata for how to display it
- Can include text, images, audio, video, applications, and more
- Can contain hyperlinks, which connect one page to another

HTML was invented in 1991 by Tim Berners-Lee, a British computer scientist, to publish his physics articles on the Internet. His physics articles required a method for delivering pages made up of text combined with images, and the ability to link the pages in a very simple manner. The linking issue was achieved by converting specific entities on the page to hyperlinks, which, when clicked, directed the user to a different page. In the same year, Tim Berners-Lee also invented HTTP as a standard for transferring the HTML pages over the Internet, a standard that is still used today for the same purpose. The combination of HTML and HTTP is the basis for today's World Wide Web.

HTML is an acronym for Hypertext Markup Language. It is not a programming language because it has no functions, variables, or flow constructs. HTML is a language that expresses the structure of a webpage. It contains data in its markup, and metadata that instructs it on how to display the data. HTML can include text, images, audio, video, forms, and many ways to express how and what to display in the webpage. It can also contain hyperlinks, which connect pages to one another and can redirect the user to other resources available on the Internet.

Working with Tags

- HTML is mostly made up of markup tags
- Tags are enclosed within angled brackets: < and >
- Most tags have a corresponding closing tag
- Each tag has its own meaning and effect

```
<div>  
  <h3> paragraph heading</h3>  
  <p>a paragraph</p>  
</div>  
<table>  
  <tr>  
    <td></td>  
    <td></td>  
  </tr>  
</table>
```

HTML is all about markup. To create an HTML page, the developer must use markup tags. Tags express HTML functionality and can enclose text or other HTML tags. To use a tag, the developer must place angled brackets (< >) around its name to first create the opening tag. For example: <head>, <body>, and <div>. Most tags have a corresponding closing tag, which is identical except for the addition of a forward slash (/) after the first angled bracket. For example: <head> and </head>. Most of the tags have a semantic, which tells the developer what is their purpose, and some tags affect their contents. For example the <p> tag indicates that its contents are a paragraph and should be rendered as a paragraph; the <table> tag indicates that its contents are a table and should be rendered as a table. There are also self-closing tags such as the image tag, which does not have a corresponding closing tag. To write self-closing tags, you put a slash (/) at the end of the opening tag. For example: or <input/>.

The HTML Document

- Also known as a webpage
- Describes how to render a webpage
- Can contain:
 - HTML tags
 - CSS blocks
 - JavaScript blocks
 - Plain text

```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>An HTML document</title>
  </head>
  <body>
    <p>Hello world!</p>
  </body>
</html>
```

The HTML document can be also be referred to as a webpage. The document describes the structure of the webpage and gives the browser all the necessary details about the elements that it will render for display. Each HTML document can contain many groups of elements, such as:

- HTML tags—These give the browser instructions about the structure and nature of the elements to be rendered.
- CSS blocks—These give the browser instructions about the style and layout that will be applied to the document. We will discuss CSS in lesson 3.
- JavaScript blocks—These give the browser instructions about the scripts that the document can run. We will discuss JavaScript in lesson 4.
- Plain text—Text blocks that are rendered as content in the document.

An HTML document is composed of the following three parts:

- An HTML element
- A Head element, which contains hidden metadata, scripts, CSS, and more
- A Body element, which contains the contents of the document

The HTML element

The HTML element is used as a container and the root of the HTML document. It wraps the head and body elements, which are mandatory elements, inside it. The main attributes that can be used in the element are `dir` and `lang`. The `dir` attribute defines the text direction inside the document, which can be right to left (rtl) or left to right (ltr). The default value of the `dir` attribute is `ltr`. The `lang` attribute specifies the base language of an element attributes or content. The language will be used by the browser for rendering purposes, such as how to render a calendar or dates.

The following example shows an HTML element.

```
<html lang="en" dir="ltr">
</html>
```

In the example, you can see that the language is set with en value, which indicates English.



Additional Reading: You can read more about language and language codes (such as en in the preceding example) in the following link: [Languages](#)

The head element

The head element is responsible for holding all the information needed to render an HTML document. It contains elements such as a title, keywords that might be useful for search engines, and other data irrelevant to the document content.

In the <head>, you will nest a title element, which represent the document title. The title element is mandatory and is displayed by the browser.

Another element that can be nested inside the head element is the meta element. The meta element is used for metadata, which is used to express data about the HTML document. The meta element can have two main attributes: charset and name.

The charset attribute specifies the character encoding used by the document. For example, it can use the utf-8 character encoding to indicate that the document is encoded by the utf-8 code page. The following example shows the meta element with the charset attribute:

```
<meta charset="utf-8" />
```



Additional Reading: You can read more about character encoding declarations in the following link: [Encoding](#)

The name attribute must be used in conjunction with the content attribute. In this case, the meta element indicates that it holds metadata with the specified name, which is its content in the content attribute.

There are standard metadata names such as application names, author names, descriptions, and keywords. This does not mean that you cannot use the meta element for your own purpose. The following example uses the meta element with the name attribute.

```
<meta name="keywords" content=" type face,font,fonts" />
<meta name="application-name" content="Contoso" />
```

There are other head elements such as link and script that are explained later in this module.

The body element

The body element is responsible for holding the content of the document. The content is rendered by the browser and displayed to the user.

The following example shows an HTML document.

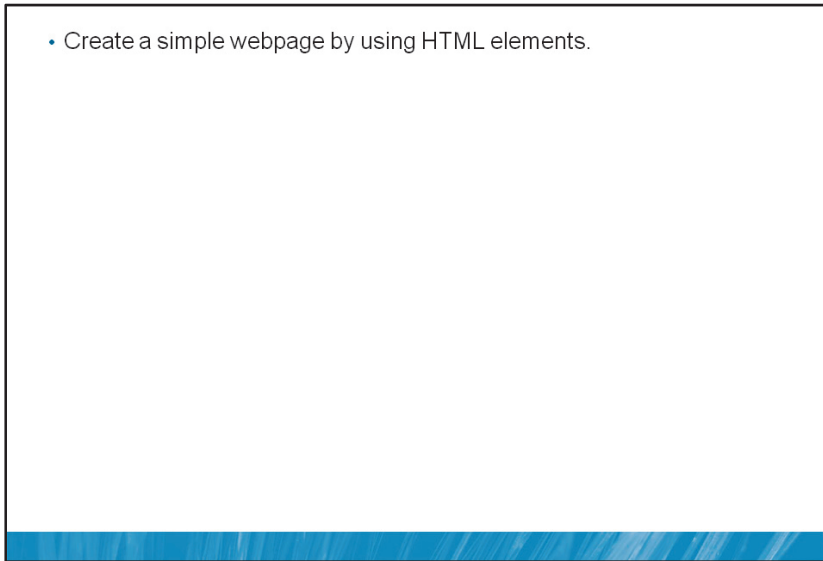
```
<!DOCTYPE HTML>
<html lang="en">
  <head>
    <meta charset="utf-8" />
    <title>An HTML document</title>
  </head>
  <body>
    <p>Hello world!</p>
  </body>
```

```
</html>
```

This example starts and ends with the HTML tag, which contains all the HTML content, including the head and body elements.

- Inside the head element, there is a title element, which is the document title that the browser will display.
- Inside the body element is content consisting of a paragraph that contains the text, "Hello world!"

Demonstration: Creating an HTML Document



This demonstration shows how to:

- Create a new Empty Web Application in Visual Studio 2010.
- Add a new HTML file.
- Add HTML elements such as table, unordered list, and hyperlinks.

Demonstration Steps

1. Log on to the virtual machine, named **SEA-DEV** as **Admin**, with the password, **Pa\$\$w0rd**.
2. Open a Notepad.exe (On the Windows **Start** menu, click **Run**, type **notepad** and then click **OK**).
3. Add the following doctype element:

```
<!DOCTYPE html>
```

4. Add the HTML element under the doctype element.

```
<!DOCTYPE html>
<html>
</html>
```

5. Create the head and body elements inside the HTML element.

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
</body>
</html>
```

6. Add a title to the head element and some content to the body element.

```
<!DOCTYPE html>
```

```
<html>
<head>
  <title>Your first HTML document</title>
</head>
<body>
  <p>Hello HTML!</p>
</body>
</html>
```

7. Save the file as an HTML file.
8. Run the saved file by double-clicking it.

Content Elements

- Content HTML elements
 - Divs
 - Spans
 - Paragraphs
- Most of the content elements hold:
 - Text
 - Other HTML elements

```
<div>This is a division.</div>
<span>This is a span.</span>
<p>This is a paragraph.</p>
```

HTML uses content elements, sometimes called, container elements. These elements are used to hold other HTML elements or text.

The Div Element

The most popular and useful content element is the div element. Div is short for division (or section). It represents a section of the HTML document and is used to group block-elements for styling purposes. When one div is placed next to another in the HTML document, they are rendered one top of the other. The use of div is the same as that of any other tag.

The following example is a division.

```
<div>This is a division.</div>
```

The Span Element

The span element is another container element, which unlike div, is used to categorize inline elements. Span does not add anything to the display, but it can wrap content and change its appearance with styling. The following example shows the span element.

```
<span>This is a span.</span>
```

The Paragraph Element

The paragraph element is used to indicate the content of a paragraph. When the paragraph is rendered, the browser will add an empty line above and below it. Paragraphs can hold most HTML elements. The following example shows a paragraph.

```
<p>This is a paragraph.</p>
```



Note: You can omit the closing tag of a paragraph and browsers will still render it correctly. Even so, it is not a good practice to omit the closing tags.

HTML Images

- Represented by using the tag
- The img tag has no closing tag
- Its main attribute is src
 - Holds the location (source) of the image to display
- Requires an alt attribute
 - Holds alternative text to display if the image is missing

```

```

A commonly used HTML element is the image. The image tag has no closing tag, unlike most other HTML elements. Instead, it links to a location where the image is stored. To create the link, the image tag must have an src attribute that contains the URL of the image.



Note: Each image element submits a request to the location of the image. This means that a webpage containing five images will submit six requests—one for the webpage and five for the images. Use images very thoughtfully because using images increases the loading time of the webpage.

If the image is unavailable or is missing, a broken link icon is displayed with alternative text, indicated by the **alt** attribute.

The following example shows the usage of an image element.

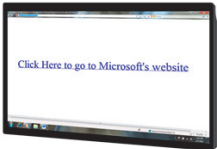
```

```

HTML Hyperlinks

- Most web pages include hyperlinks to other web pages
- Hyperlinks are represented by the `<a>` tag
- Links allow you to:
 - Navigate from one page to another
 - Create a bookmark inside the web page
- The main attribute of the `<a>` element is `href`

```
<a href="http://www.microsoft.com">Click Here to go to Microsoft's Website</a>
```



One of the most important features of the HTML specifications is the ability to connect webpages and browse between one page and another. This ability was introduced very early in the development of HTML because it helped to categorize relevant content into a full website or application. This was achieved by using hyperlinks—HTML elements written into a webpage mainly to navigate from one page to another or to create a bookmark inside the page. The `<a>` tag, when it wraps text or images, makes them into hyperlinks. When a pointer is moved over a hyperlink, the pointer arrow changes into a small hand, indicating that this is a clickable part of the webpage.

The following example shows how to create a link.

```
<a href="some URL">a link</a>
```

As seen in the example, the HYPERLINK element uses the **a href** attribute, indicating the destination of the hyper reference.

Another important attribute is the **target** attribute. This attribute specifies where to open the linked page. For example the use of **_blank** will open a new window for the clicked link. The **target** attribute can get the following values:

- **_blank**—open the link in a new window or tab.
- **_self**—open the link in the same window or tab. This is the default value.
- **_parent**—open the link in its parent frame.
- **_top**—open the link in the full body of the window.

Two other attributes are the **name** and **id** attributes. When these attributes are used inside the link, they create an anchor behavior. The anchor behavior is a synonym for creating a bookmark in a page section. In the following example, the first link is a bookmark, and the second link takes the user to the bookmark when it is clicked.

```
<a id="bookmark">This is a bookmark</a>
<a name="bookmark">This is a bookmark</a>
```



```
<a href="#bookmark">Visit the bookmark</a>
```

Note that the link for visiting the bookmark has a # prefix, which indicates to the browser to search for a bookmark with the name or id of "**bookmark**".

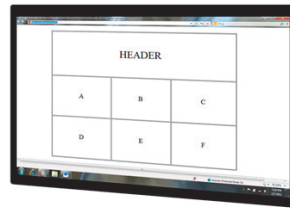


Note: The bookmark/anchor behavior is used mainly to create the "Table of Contents" section at the beginning of a very long document.

HTML Tables

- Main purpose is to organize tabular data
- Represented by the `<table>` element, which includes:
 - `<tr>` elements to represent table rows
 - `<td>` elements to represent table cells
 - `<th>` elements to represent table headers
- Sometimes misused to create webpage layouts

```
<table>
  <tr>
    <td>row 1, cell 1</td>
    <td>row 1, cell 2</td>
  </tr>
  <tr>
    <td>row 2, cell 1</td>
    <td>row 2, cell 2</td>
  </tr>
</table>
```



The HTML table helps to organize tabular data inside a table structure. It is represented by the `<table>` element, which is a container for table rows. Table rows, `<tr>` elements, are divided into table cells (`<td>` elements). Every table cell (or **table data**) can contain HTML elements such as images, links, other tables, and more.

The following example shows a table.

Table Element

```
<table>
  <tr>
    <td>row 1, cell 1</td>
    <td>row 1, cell 2</td>
  </tr>
  <tr>
    <td>row 2, cell 1</td>
    <td>row 2, cell 2</td>
  </tr>
</table>
```

To display a border around the table, a **border** attribute with a width value in pixels should be added to the table. For example, the following table will have a 2 pixel border:

```
<table border="2">
</table>
```

Tables can also contain table headers, that is, `<th>` elements. Table headers are displayed in bold style and centered by most of the browsers. Tables can be written without table headers.

The following example shows a table with table headers.

Table with Table Headers

```
<table>
  <tr>
```

```
<th>header 1</th>
<th>header 2</th>
</tr>
<tr>
  <td>row 1, cell 1</td>
  <td>row 1, cell 2</td>
</tr>
<tr>
  <td>row 2, cell 1</td>
  <td>row 2, cell 2</td>
</tr>
</table>
```



Note: The W3C specification for tables states that “Tables should not be used purely as a means to layout document content”. In the past, tables were used for page layout, but this is very unpopular, and as written in the specification, it is not recommended because of the following reasons:

Tables are rendered only after the browser reaches the closing tag, `</table>`. This is because up to that point, the browser cannot know how many columns the table will have. This behavior slows down the rendering, because the browser cannot display the first part of the webpage while reading the rest.

- Tables create a lot of distraction for developers. Because tables have a lot of structural elements, the pages they create are often heavier and can sometimes be unreadable.
- The use of CSS styles is better for page layout and organization.

HTML Lists

- Divided into three main lists:
 - Unordered list
 - Ordered list
 - Definition list
- Represented by , , and <dl> respectively
- Can have one or more list items
- Very popular to use in menus

```
<ol>
  <li>Item1</li>
  <li>Item2</li>
</ol>
```



There are three kinds of lists in HTML:

- Unordered lists ()
- Ordered lists ()
- Definition lists (<dl>)

Every list element can hold one or more list items ().

The following example shows an unordered list.

Unordered List

```
<ul>
  <li>Item1</li>
  <li>Item2</li>
</ul>
```

The preceding code is rendered as a bulleted list:

- Item1
- Item2

The following example shows an ordered list.

Ordered List

```
<ol>
  <li>Item1</li>
  <li>Item2</li>
</ol>
```

The preceding code is rendered with numbers, instead of bullets:

1. Item1

2. Item2

List items can hold any kind of HTML elements, even other lists (which will create a behavior of a nested list).



Note: It is very common to use unordered lists as menus and list items as its menu items.

Definition lists are a little bit different and rarely used. These lists do not use the `<li>` element; they use the `<dt>` and `<dd>` elements instead. The `<dt>` element defines the item inside a definition list, and the `<dd>` element describes the item.

The following example shows how to use the definition list.

```
<dl>
  <dt>Coffee</dt>
  <dd>- black hot drink</dd>
  <dt>Cola</dt>
  <dd>- black cold drink</dd>
</dl>
```

The preceding code is rendered as follows:

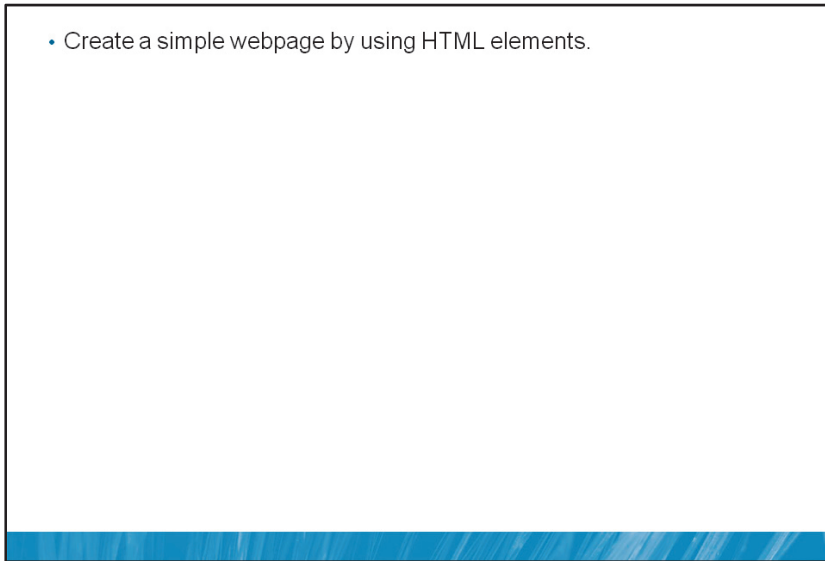
Coffee

- black hot drink

Cola

- black cold drink

Demonstration: Creating an HTML Document with HTML Elements



This demonstration shows how to:

- Create a new Empty Web Application in Visual Studio 2010
- Add a new HTML file.
- Add HTML elements such as table, unordered list, and hyperlinks.

Demonstration Steps

1. Log on to the virtual machine named **SEA-DEV** as **Admin**, with the password. **Pa\$\$w0rd**
2. Open Visual Studio 2010.
3. Create a new C# ASP.NET Empty Web Application.
4. Create an HTML file.
5. In the HTML file's body element, add the following elements.

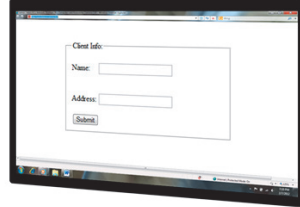
```
<div>This is a division.</div>
<div> This is another division</div>
<span>This is a span</span>
<span>This is another span</span>
<p>This is a paragraph</p>
<a href="http://www.microsoft.com">A link to Microsoft site</a>
<table border="1">
  <tr>
    <th>header 1</th>
    <th>header 2</th>
  </tr>
  <tr>
    <td>row 1, cell 1</td>
    <td>row 1, cell 2</td>
  </tr>
  <tr>
    <td>row 2, cell 1</td>
    <td>row 2, cell 2</td>
  </tr>
```

```
</table>
<ul>
  <li>item1</li>
  <li>item2</li>
</ul>
```

6. Save the file as an HTML file.
7. Run the saved file and view the completed webpage.

Creating Forms

- The best way to pass data from client to server
- Can contain form elements such as
 - Input elements
 - Select lists
 - Buttons
- Wrapped inside a `<form>` tag



```
<form action="#" method="GET">
  <fieldset>
    <legend>Client Info:</legend>
    <label for="txtflame">Name:</label><input type="text" id="txtflame" name="txtflame" /><br />
    <label for="txtflame">Address:</label><input type="text" id="txtflame" name="txtflame" /><br />
    <input type="submit" value="Submit" name="btnSubmit" />
  </fieldset>
</form>
```

Most of the applications on the Internet have a server-side that handles requests and sends responses back to the client. When there is a need to pass data from the client side to the server, one way to do it is by means of a form. A form defines a collection of input items that hold user-defined values. These items can include elements such as text boxes, check boxes, and radio buttons. All such input elements must be enclosed inside a `<form>` tag, which indicate that they are part of the form and that their values are sent back to the server. The form needs to include a means of submission (for example, a **Submit** button), which submits the entire set of values when activated by the user.

The Form Tag

A form is constructed from the input elements defined between its opening and closing tags. All values obtained from these input elements are sent when the form is submitted by the user. The main attributes of the form include:

- **Action**—a destination URL to which the form will submit its information. The default value for the attribute is the URL of the current document. The following example shows a form.

```
<form action="http://someurl">...</form>
```

- **Method**—the method by which the form is submitted. There are two options: GET and POST. GET (or HTTP GET) is a method used to transmit an HTTP request with the GET verb. All the names and values of the input element are appended as a query string to the URL specified in the action attribute.



Note: Query string is data that is appended to the end of a page URL. They are created by adding a question mark to the end of the URL and then a name=value pair for each name and a value separated by comma. Here is an example of shows a query string:
http://www.microsoft.com?a=1;b=2. In the example, there are two parameters in the query string – a and b string—a and b with their values.

- POST (or HTTP POST) is a method used to transmit an HTTP request with the POST verb. All the names and values of the input elements are sent in the request header, which is a header of an HTTP

request, to the location specified in the **action** attribute. POST is preferred because it does not send bare data like GET does. The following example shows a form tag that uses the POST method.

```
<form action="http://someurl" method="POST">...</form>
```

The Input Elements

There are many types of input elements. Input elements are defined between the form tags. To use an **input** element, you must use an **input** tag and specify the type of the input. If there is no type, or the browser cannot recognize the type, a textbox input type is used as the default.

Each input type must have a unique name. This restriction is necessary because when a form is submitted, the name of the input is sent to the server with its data.

The following input types are the most common:

Text fields: Input with the **text** type is rendered as a one-line text field. Text fields are the most common input types. The text field can receive any value that the user types inside it. If the **value** attribute is set, the text field is rendered with the given string as a default value.

```
<input type="text" name="txtFirstName" value="John" />
```

Radio buttons: Radio buttons let the user select only one option from a limited number of choices. The input type needs to be **radio** to create a radio button. To categorize radio buttons and indicate that only one of the buttons in the group can be selected, the developer must give them all the same value under the **name** attribute. A **value** attribute must appear in each radio button, defining the text that will appear beside the radio button when it is rendered. At form submission, the list of name/value pairs sent to the server will include a single pair representing the entire group of radio buttons. In that pair, the name is the mutual name of the buttons, and the value is that of the selected radio button. The **checked** attribute may be added to one radio button from each group, to indicate that it should be checked by default when the page is loaded. The following example shows two radio buttons.

```
<input type="radio" name="rdo" value="Yes" />
<input type="radio" name="rdo" value="No" />
```

Check boxes: Check boxes let the user select options from a limited number of choices. The input type needs to be a **checkbox** to create a check box. To show text beside the check box, a **value** attribute must be supplied. When the form is submitted, the list of name/value pairs sent to the server includes a name and value pair for each selected check box. The **checked** attribute may be used to designate that the check box should be selected by default when the page is loaded. The following example shows two sample checkboxes.

```
<input type="checkbox" name="chkYes" value="Milk" />
<input type="checkbox" name="chkNo" value="Cola" />
```

Submit buttons: Submit buttons are created with the **submit** input type. The browser will render a button, which when clicked, constructs the list of name/value pairs from all the input types in the same form, and sends it to the server. The following example shows a **Submit** button.

```
<input type="submit" name="btnSubmit" id="btnSubmit" value="Click Me" />
```

Password field: The password field has the same behavior as a text field, but displays only asterisks, instead of the text that is input. The following example shows a password field.

```
<input type="password" name="txtPassword" />
```

Buttons: This input type appears as a button on the webpage. Each button must have a **value** attribute that holds the text that is displayed on the button surface. Buttons have no effect on form submission and have no functionality by default. To add behavior, an event handler must be wired to the button by using JavaScript. JavaScript is explained in lesson 4. The following example shows a button.

```
<input type="button" id="btnClick" name="btnClick" value="Click Me" />
```

Hidden field: The hidden field is a text box that is not displayed to the user. The hidden field value is not intended to be manipulated by the user. The following example shows a hidden field.

```
<input type="hidden" id="txtHidden" name="txtHidden" value="not displayed to the user" />
```

File input type: The file input type represents a list of selected file names. It is rendered as a text box followed by a browse button. Pressing the browse button opens a dialog box that enables the user to choose files to upload and send to the server. The files are not uploaded until the form is submitted. The following example shows a file input type.

```
<input type="file" id="fileUpload" name="fileUpload" />
```

Reset button: Reset buttons are created with the reset input type. When a reset button is clicked, it resets all the form input types to their default state. You can use the **value** attribute to write the text that is displayed on the button surface, just as you do in all the other button input types. The following example shows a reset button.

```
<input type="reset" id="btnReset" name="btnReset" value="Reset" />
```

Labels

Labels define a label for an input type. Labels have no effect on their shown content. However, they provide usability improvement for mouse users because they will toggle the control when clicked. Labels should use the attributes that need to hold the id of the input type related to the label. The following example shows a label and an input type.

```
<label for="txtFirstName">First Name: </label><input type="text" id="txtFirstName" name="txtFirstName" />
```

Drop-Down Lists

Drop-down lists are created by the **select** element. They instruct the browser to render a drop-down list that displays available options when it is clicked. Each of the options is created by an **option** element.

The following example shows a drop-down list.

Drop-down list

```
<select>
  <option value="0">Male</option>
  <option value="1">Female</option>
</select>
```

Two attributes that you should get to know with drop-down lists are the **size** and **multiple** attributes.

The **size** attribute gets a number that defines the amount of visible options in the drop-down list. If the size attribute has a value greater than 1, the drop-down list will look like a list box instead. The default value of the size attribute is 1.

The **multiple** attribute is used to indicate that multiple options can be selected in the drop-down list.

Fieldsets and Legends

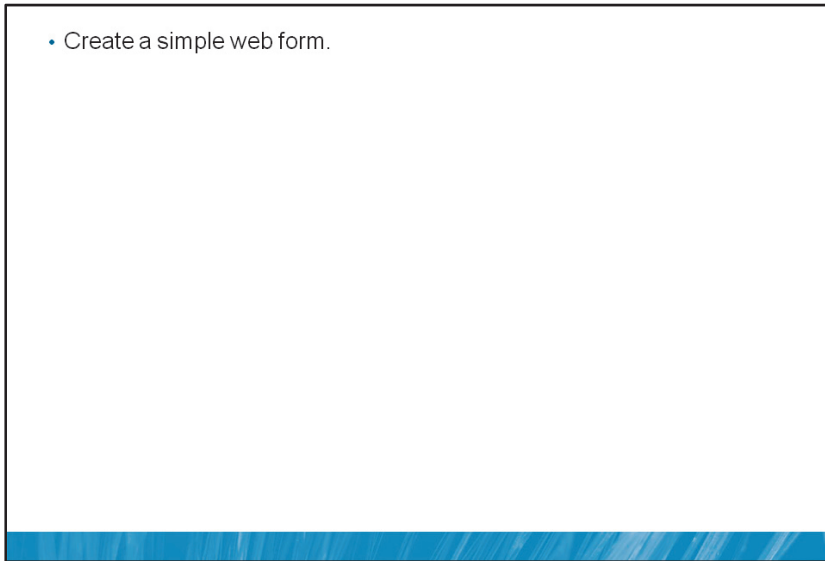
Fieldsets are used to logically group form elements. Fieldsets will also create a border around all the fields of a form. A legend can be used to define a caption for a fieldset.

The following example shows a fieldset.

Fieldset

```
<fieldset>
  <legend>Client Info:</legend>
  <label for="txtName">Name: </label><input type="text" id="txtName" name="txtName"
/><br />
  <label for="txtAddress">Address: </label><input type="text" id="txtAddress"
name="txtAddress" />
</fieldset>
```

Demonstration: Creating Forms by Using HTML



This demonstration shows how to:

- Create a new Empty Web Application in Visual Studio 2010.
- Add a new HTML file.
- Add a form element.
- Add a fieldset and legend elements.
- Add input types and labels.

Demonstration Steps

1. Log on to the virtual machine named **SEA-DEV** as **Admin** with the password **Pa\$\$w0rd**
2. Open Visual Studio 2010.
3. Create a new C# ASP .NET Empty Web Application.
4. Create an HTML file.
5. In the HTML file's body element, add the following element.

```
<form action="#" method="GET">
  <fieldset>
    <legend>Client Info:</legend>
    <label for="txtName">Name: </label><input type="text" id="txtName"
name="txtName" /><br />
    <label for="txtAddress">Address: </label><input type="text" id="txtAddress"
name="txtAddress" /><br/>
    <input type="submit" value="Submit" name="btnSubmit" name="btnSubmit" />
  </fieldset>
</form>
```

6. Save the file as an HTML file.
7. Run the saved file and view the completed webpage.

Lesson 2

What's New in HTML5?

- What Is HTML5?
- The Doctype Element
- The New Elements in HTML5
- HTML5 and the Open Web Platform

HTML5 allows for the creation of better web applications and websites. Using HTML5, web developers can expose graphics and multimedia to webpages with no need for plugins, use advanced JavaScript APIs such as client-side storages, arrange their webpages with a more meaningful structure, and embed semantics in their content. This lesson explains the past, present, and future of HTML5.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain what HTML5 is.
- Explain the new HTML5 elements (structural elements, new form/input elements, new attributes, new media content, etc.).
- Describe the doctype element.
- Describe the Open web platform.

What Is HTML5?

- The new version of HTML
- A new and emerging set of web standards and specifications
- HTML5 is HTML + CSS3 + JavaScript APIs



In 1991, the first HTML specifications were published.

The second release of HTML (HTML2) was published in 1995, introducing form-based file uploads, tables, and more.

In early 1997, a new and improved HTML specification (HTML3) was published as a W3C recommendation.



Note: Most of the changes in this version were the adoption of the visual markup tags of Netscape.

In late 1997, the last HTML specification (HTML4) was released. This version has changed a little over time, but remained in use until 2007, when the W3C and WHATWG began work on HTML5.



Note: WHATWG is the Web Hypertext Application Technology Working Group. It is a working group separated from W3C, and its main issue is web forms.

HTML5 is a product of two different groups that were working on separate, but related, efforts:

- W3C—was working on HTML4.
- WHATWG—was working on web forms and web applications.

These groups began to cooperate in 2006 to create a better web experience. The product of their cooperation is HTML5. HTML5 is a new set of standards and specifications that try to fix some problems with the previous version of HTML, such as the lack of guidelines, the loose structure, the lack of flexibility, and more.

The Doctype Element

- A declaration of document type
- Must be provided by any HTML document
- Defines the rendering mode of the browser
- Was created to enable HTML parsing and validation

```
<!doctype html>
```

The doctype is one of the important elements in any HTML document. It is the first element that you include in an HTML document. It is used for defining the document type declaration (DTD), which defines to the browsers how to render the HTML document (this is called the Standards Mode). At the beginning, it was created to enable HTML parsing and validation by third-party SGML tools. Current browsers do not render HTML as SGML applications and do not read the DTD, but **doctype** is mainly used for defining quirks mode. *Quirks mode* is a technique that browsers use for backward compatibility. Because there are many browser versions (for example, Internet Explorer 6, Internet Explorer 7, Internet Explorer 8, and Internet Explorer 9), most of the browsers maintain a mode that enables them to render webpages that were created for older browsers. This ability is called the quirks mode. In HTML5, the **doctype** element was simplified into the following code.

```
<!doctype html>
```

This element does not hold any DTD. The following example shows the use of DTD in HTML4.

```
<!doctype html public "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
```

The New Elements in HTML5

- The **canvas** element
- The **video** and **audio** elements
- New content-specific elements:
 - **article**, **footer**, **header**, **nav**, **section**, and more
- New form controls:
 - **calendar**, **date**, **time**, **email**, **url**, **search**, and more

HTML5 includes many new elements and features. There are approximately 100 different working groups that write its specifications, and this mass of groups produces a lot of changes. The following list describes some of the new elements of HTML5:

- **Canvas**—A drawing surface that enables drawing 2D graphics with JavaScript API.
- **Video** and **Audio**—elements that enable the inclusion of multimedia in webpages without the need for plug-ins such as Flash. They also expose a JavaScript application programming interface (API) that allows you to interact with them programmatically.
- New content-specific elements—Semantic elements such as **article**, **footer**, **header**, and **nav** enable the developer to organize content with more expressive elements.
- New form controls—New types of input elements were introduced. These elements make it the browser's responsibility to implement the behavior of calendars, dates, and email addresses, freeing developers from having to create the behavior themselves.

These new elements come with many new JavaScript APIs that enable browser functionality that was not available before HTML5. This includes determining the location of the browser by using the Geolocation API, the use of files through the File API, and more.

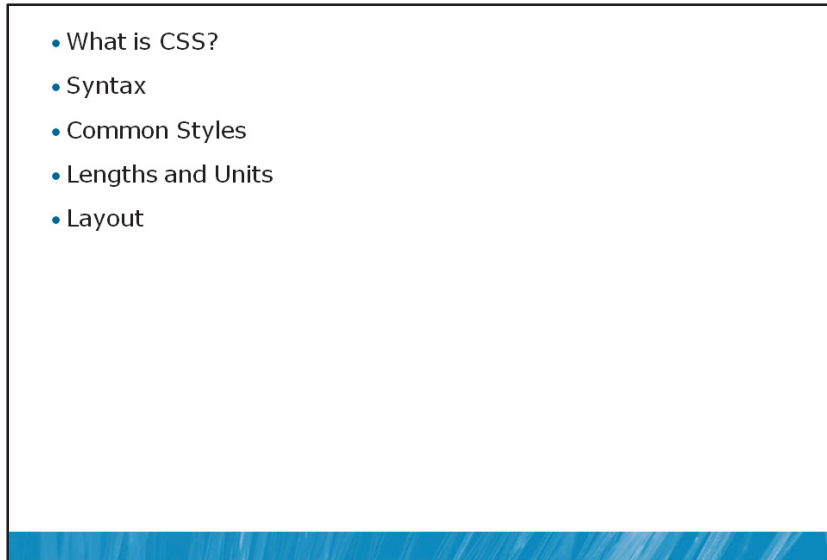
HTML5 and the Open Web Platform

- The Open Web Platform (OWP) is a collection of web technologies
- It is developed by W3C and other web standardization bodies
- HTML5 is part of the OWP

The open web platform, or OWP, is a collection of web technologies and standards developed by W3C and other web standardization bodies. OWP is an umbrella term that W3C defined to cover standards such as HTML5, CSS3, SVG, and most of ECMAScript APIs throughout 2011. W3C uses this term in an attempt to differentiate between this set of technologies (which many refer to as HTML5) and HTML5 proper (which is technically a single, large standard written by W3C).

Lesson 3

Introduction to CSS



CSS is a language with a single purpose: to tell the browser how to display webpages. All modern browsers support CSS, and most modern websites use it. CSS can be used to define almost every aspect of a website presentation, other than the content itself— fonts, colors, general layout, and a consistent look and feel. CSS is integral to every professional web developer's tool chest.

In this lesson, you will learn the CSS basics that will enable you to integrate CSS into your webpages. You will also learn about some of the limitations of CSS and ways to resolve issues.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe CSS in general.
- Explain the CSS syntax.
- Explain common CSS styles.
- Explain the CSS approach to units and lengths.
- Describe what is layout and how to use CSS to create a layout.

What Is CSS?

- CSS defines styles and layout for HTML elements
- CSS separates the page content (HTML) from its presentation (CSS styles)
- CSS styles can be declared:
 - Inline
 - With the **style** element
 - In CSS files referenced with the **link** element
- CSS styles *cascade*
- Visual Studio has style dialog boxes, property windows, and IntelliSense support
- Working with CSS takes a lot of trial and error
- Browser Compatibility
 - Progressive Enhancement
 - Graceful Fallback

Separation of Concerns: Page Content vs. Presentation

HTML is a *markup language* that ascribes meaning to content by wrapping it with descriptive tags. CSS, on the other hand, is a *declarative language* that defines the styles and layout that a browser should apply to HTML elements matching specific conditions. There is a very clear *separation of concerns*: HTML is about content, data and information, while CSS is about presentation.

CSS stands for *Cascading Style Sheets*. The term *cascade* simply means that styles *accumulate*. Every rule that matches an HTML element adds more style information that the browser can use to draw or lay out that element. Properly designed websites often contain multiple CSS files, or *style sheets*. It is not uncommon for multiple rules across all of these style sheets to match any given element. In fact, in addition to the style sheets that a website uses, every browser supplies a default style sheet and users may supply their own also.



Best Practice: Using multiple style sheets, where each style sheet targets a specific aspect of the website, is strongly recommended. For example, one style sheet may supply styles for the general theme and another may supply styles for a specific feature, like a cart or a menu. Separating these styles into multiple files makes it easier to manage the styles and maintain them over time.

Thus, style sheets may overlap. In other words, any number of rules can match any given element, and these might declare conflicting styles. For example, one rule might specify that all the text on the page is 24 pixels tall, while another rule might specify that headers should be 36 pixels tall. When the browser searches for all the styles that match a given header, it has to choose between the two sizes. The CSS standard defines very clear rules of precedence that browsers (and therefore, web developers) must follow when they create the CSS.

Basically, specificity (or closeness) matters. The general rule of thumb is that the more specific, or closer, a rule is to an element, the more likely it is to take precedence in such a conflict. If all the style sheets were

to be laid out in the order they are declared, the rules literally *closest* to the matching element are more likely to be applied.

Declaration

There are a few places where styles can be declared:

- Inline styles are declared in the **style** attribute of the **HTML** tag. They are as close (technically, as “specific”) as they can possibly be, and therefore, they always take precedence.

The following example shows an inline style applied to a p (paragraph) element.

Inline Styles

```
<p style="color: red;">This is a red paragraph.</p>
```

- Styles declared in the **style** element in the **head** element apply only to elements on the page.

The following example demonstrates the use of a style declared in the style element that affects every p element on the page.

The Style Element

```
<html>
<head>
  <style>p { color: green; }</style>
</head>
<body>
  <p>This is a green paragraph.</p>
</body>
</html>
```

- Styles declared in external CSS files can be referenced with the **link** tag and applied to the elements on every page that references them.

The following example references a CSS style sheet by using the link element.

Using the link Element

```
<head>
  <link rel="stylesheet" href="styles.css" />
</head>
```

Visual Studio Support

Although professional web developers tend to prefer writing their CSS styles manually, Visual Studio offers a number of very helpful aids that assist both new and experienced developers, including the CSS Properties window, the **New Style** dialog box, the **Build Style** dialog box, and IntelliSense.

The Visual Studio **Build Style** dialog box is an especially handy tool to find and edit many of the common CSS styles. Unlike many other CSS editors, it does not introduce extraneous styles and avoids changing existing code where possible. It also produces clean CSS. This makes it one of the most intuitive CSS editors, and its use is encouraged for new web developers.

As programmers gain experience, they learn to rely on IntelliSense for code completion and style discovery, though the **Build Style** dialog box remains a helpful tool.

Trial and Error

Editing website CSS is very much an exercise in trial and error. Making a professional site that looks good takes time and effort, and is a gradual, iterative process. However, there are excellent tools available to facilitate this process, including the Internet Explorer Developer Tools that are bundled with Internet Explorer, starting with Internet Explorer 8. These tools let web developers experiment with the styles on the page before copying their results to the style sheet files.

Browser Compatibility

Another important thing to understand is browser compatibility. CSS is an ever-evolving standard. Much of the standard was developed in parallel to its implementation by browser vendors. So, different browsers have varying levels of compatibility with the standard; and on occasion, they have been known to interpret the standard differently. Newer browsers are more compliant and compatible with each other than before, but differences still exist.



Note: Web developers should always check to ensure that their sites work properly on all the browsers they wish to support.

Two common approaches that web developers use to ensure at least minimal compatibility are *progressive enhancement* and *graceful fallback*. The two approaches are very similar conceptually, but are implemented differently. Web developers who progressively enhance their site tend to build the site with the minimal target browsers in mind, and then add support for more advanced features targeted to newer browsers. Web developers who follow the graceful fallback approach tend to target the newest browsers first, and use various solutions and tricks to ensure that the site works properly on older browsers they wish to target.

Syntax

- Basic Syntax
- Selectors
- Properties and Values
- Shorthand Properties and Default Values
- Naming Conventions and Case-Sensitivity
- By Tag Name
- By Class Name
- By Element ID
- Grouping Selectors
- Combining Selectors (div.class1.class2)
- Descendant and Child Selectors
- Pseudo-classes
 - a:link, a:visited, a:hover, and a:active
 - :first-child
- Pseudo-elements
 - :first-letter and :first-line
 - :before and :after

```
selector {
  property: value;
}
```

Basic Syntax

A CSS rule is composed of a selector and a set of style declarations. The selector is a kind of query that the browser uses to find the HTML elements that the styles in the rule should be applied to. Each style declaration consists of a property name and a value separated by a colon (:). The semicolon (;) ends the declaration. Multiple style declarations can be specified in a single rule.

CSS have the following basic structure.

The Structure of a CSS Rule

```
selector {
  property: value;
}
```

Selector Syntax

HTML is a hierarchical document structure. This means that every element has certain hierarchical relationships with its surrounding elements. An element may have ancestors, descendants and siblings. CSS selectors describe certain relationships that have to exist for a style to be applied to an element.

The following example demonstrates a simple rule that applies a couple of styles to the TH element.

Simple CSS Rule

```
th
{
  font-size: large;
  background-color: orange;
}
```

This example contains a very simple CSS selector that instructs the browser to find all the **TH** (table header cell) elements on the page, regardless of their position in the hierarchy. It then paints those elements with a background orange color and sets the font to a large size. Selectors are explained in more detail later.

Properties and Values

CSS has a large but finite list of properties. Every property has very specific, well-defined behavior. Also, every property can have only certain kinds of values. The names and values follow certain conventions, so they are usually easy to remember. Visual Studio supports IntelliSense and code completion for CSS properties and their values, so finding (or discovering) properties and recalling their specific constraints is easy.



Note: No single browser recognizes all the CSS properties. Even browsers that recognize the same properties might not render them quite the same way. The situation has improved in recent years and most modern browsers interpret the same basic set list of CSS properties the same way. However, the CSS standard is always in flux and new properties or refinements of existing values are being introduced all the time. In the interim period before a property becomes standardized and is implemented more or less the same way by all the major browsers, each browser vendor may introduce their own properties. These special browser-specific properties are always prefixed with a “vendor prefix”. For example, a new CSS property that only Internet Explorer supports might be called, `-ms-new-property`. We recommend that you avoid these vendor-specific properties unless you plan to maintain the site actively, because the behavior might change unexpectedly.

CSS properties can very often accept multiple kinds of values. For example, the **color** property value can be specified as a named color, such as **orange** or **blue**, or as a hexadecimal color value, such as **#3bf**. Another example that highlights the flexibility of using CSS is size units. Sizes, which are values that can be specified for many properties, including **width** and **height**, can be specified in absolute point or pixel values or in relative percentage values.

CSS always tries to “fill in the blank”. Every property has a default value. Web developers only specify the minimal set of properties and values that they need to achieve the design they want.

Shorthand Properties and Default Values

There are a large number of CSS properties, but they are generally arranged by groups. For example, there are properties that affect the font, so they are all named with a **font-** prefix (for example, font-size, font-weight, font-family, etc.). These prefixed fonts are called longhand properties. Longhand properties allow web developers to declare very specific styles in a verbose manner.

Sometimes that verbosity is unneeded, so shorthand properties can help. Shorthand properties are shorter properties that can specify the behavior for multiple longhand properties belonging to a single group in a single terse declaration (for example, **font**).

Shorthand properties have a number of advantages:

- They consume less bandwidth than their longhand equivalents.
- They are considered “cleaner” and more elegant.
- They are easier to remember and write.

Shorthand properties unite a number of related properties into one declaration. To facilitate even shorter declarations, the values can be abbreviated also.

The following example shows longhand properties.

Margin Specified by Using Longhand Properties

```
margin-top: 10px;
```

```
margin-right: 10px;
margin-bottom: 10px;
margin-left: 10px;
```

The following example shows the preceding code rewritten by using a single shorthand property.

Margin Specified in Full with the Shorthand Property

```
margin: 10px 10px 10px 10px;
```

The code can be shortened further by removing redundant sizes, as the following example shows.

Margin Specified in Most Abbreviate Form

```
margin: 10px;
```

The property in the last declaration is shorthand for the four explicit properties. Some of the Developer Tools extensions will only show the expanded, longhand properties, even when the style sheet declares a shorthand property.

The size in the preceding declaration is just one example. It is possible to specify 1, 2, 3, or 4 of the sizes. If only one size is specified, it applies to all the sides; if two or three are specified, they apply to the margins, in order, beginning from the top in a clockwise direction. Margins left unspecified are set equal to those on the opposite side.

The same kind of shortening can be used for many other properties also, including font, border, and padding.

Naming Conventions and Case-Sensitivity

CSS should be written as though it is case-sensitive. The CSS standard specifies that CSS is not case-sensitive, but not all browsers interpret this in the same way. For example, some browsers rely on case-sensitivity for pages that use the XHTML schema. In addition, external resources referenced by CSS styles may require case-sensitivity in all browsers.



Best Practice: Because different browsers interpret the case-sensitivity rules differently, web developers should always write CSS as though it is case-sensitive. They should also follow common naming conventions.

The following list describes the most common naming convention rules:

- HTML tag names in selectors and in HTML documents should be in lowercase (for instance, **h1**, **table**, **a**).
- Class names should use lowercase (for instance, **.notice**, **.footer**, **.menu**).
- Compound class names (class names with multiple words) should have those words connected with hyphens (for instance, **.page-link**, **.article-body**, **.very-important-notice**).
- Element IDs should use lower camel case (for instance, **#logo**, **#mainNavigationMenu**, **#errorSummary**).
- Class names and element IDs should be descriptive and easy to understand.

Selectors

An HTML document is a hierarchical document that the browser represents as a tree of elements. Selectors are expressions that describe hierarchical patterns, which are essentially relationships between the elements in the tree. When the browser encounters a style rule, it finds all the elements whose positions in the tree are described by the pattern expressed by the selector. The browser then applies the styles declared in that rule to those matching elements.

By Tag Name

The simplest pattern is a Type Selector, which is an expression that matches a tag name. For example, to instruct the browser that the text in every paragraph element (denoted by the `<p>` tag) in the page should be green, the selector would simply be the tag **p**. (According to common naming conventions, tag selectors should use lowercase.)

The following example shows how to use the whole style rule.

A Type Selector

```
p { color: green; }
```

By Class Name

Very often, web developers may want to apply the same style to a number of elements with different tags. For example, a web developer might decide that important notices should use the color, red, whether they are in paragraphs, headers, or links. Instead of applying the red color to all the **p**, **a**, and **h1** through **h6** tags, the developer would instead choose a name for the style. This name should be in lowercase, preceded by a period (.), and should be hyphenated if it is more than one word. This style will then be implemented by referencing its name with the **class** attribute anywhere in the HTML where it is to be used. Classes are vital for separating the UI from the content, because they enable the use of agnostic styles that can be applied to an element by intention, role, or semantic relationship, instead of an arbitrary hierarchical structure.

The following example shows how to define the style.

A Class Selector

```
.important-notice { color: red; }
```

The following example shows the matched HTML fragment.

HTML Matched by a Simple Class Selector

```
<p>This paragraph is green because all paragraphs are green.</p>  
<p class="important-notice">But this paragraph is red because it uses the "important-notice" class name.</p>
```

By Element ID

In contrast to classes, element ID selectors match only the element with the specified ID. The ID of an element is specified by using the **id** attribute, and must be unique in the HTML document. It is a violation of the HTML standard to use the same ID on more than one element in a page. Otherwise, the ID selector is much the same as the class selector because it can match any element, regardless of its tag. ID selectors are often used for very specific elements that are repeated on multiple pages, including navigation menus and specific sections (such as page headers and footers, or the page logo). They are generally faster, but their use is discouraged for more general scenarios because they limit the flexibility of the page design.

Element IDs are very useful when working with JavaScript. By convention, element IDs should be camel-cased in accordance with JavaScript naming conventions.

The following example uses the purple color for the page header.

An Element ID Selector

```
#pageHeader { color: purple; }
```

The following example contains an HTML fragment matched by the previous CSS selector.

An HTML Fragment Matched by an Element ID Selector

```
<h1 id="pageHeader">This is the Purple Page Header</h1>
```

Grouping Selectors

It is often convenient to apply a common style declaration to multiple selectors. For example, the designers might decide that both the page header and the navigation links should be purple. We strongly discourage creating class names for artificial relationships based on common styles. For example, a class name called, "purple", which could be used on both the page header and the navigation links, would violate the separation of concerns. If the page design requires a different color, the web developer would be inclined to modify both the CSS and the HTML. Our goal is to create a resilient separation that would not require modifying the HTML for stylistic changes. A better solution is to group different selectors in one style rule. Changing the color would require modifying only the CSS. Grouped selectors are separated by a comma, as shown in the following example.

```
.nav-link, #pageHeader { color: purple; }
```

Combining Selectors

A single selector can also combine multiple conditions. For example, a designer might decide that important notices that appear in regular paragraphs should be red, but that important notices that appear in table cells should be orange. Instead of creating two different classes for what are essentially and semantically, identical, the web developer need only modify the CSS. To create combined selectors, the selectors need to be written adjacent to each other, with no spaces in between.

```
p.important-notice { color: red; }
td.important-notice { color: orange; }
```

These selectors can be read as logical AND conditions. For example, apply the red color style to all elements that have *both* the "p" tag AND the "important-notice" class. More conditions can be affixed. The following example specifies that all elements with the "menu-link" class name should be blue, except for elements that have the "a" tag AND the "menu-link" class AND the "current" class. The class attribute can specify multiple classes by separating them with spaces.

```
/* CSS */
.menu-link { color: blue; } a.menu-link.current { color: black; }
<!-- HTML -->
<a href="thispage.html" class="menu-link current">This page</a><a
href="thatpage.html" class="menu-link">Some other page</a>
```

Descendant and Child Selectors

Selectors are most powerful when they describe more complex relationships between elements. A selector can be composed of multiple selectors separated by various "combinators". Combinators define the

relationships between the different selectors. The simplest combinatory is a space, which denotes a “descendent” relationship. The pattern “.menu a”, for example, matches every anchor element (denoted by the “a” tag) that is a descendent of an element with the menu class name. Consider the following CSS rule.

```
.menu a { color: yellow; }
```

This rule would match the first “a” tag because it is a child (an immediate descendent) of the div with the menu class, the second “a” tag because it is also a descendent of the “menu” div, but not the third “a” tag, because it is a sibling of the “menu” div:

```
<div class="menu">
  <a href="Page1.html">Page 1: yellow</a>
  <ul>
    <li><a href="Page2.html">Page 2: yellow</a></li>
  </ul>
</div>
<a href="Page3.html">Page 3: not yellow</a>
```

If the selector used a child combinator, denoted by a greater than symbol (“>”), it would only match the first “a” tag. The second “a” tag is not an immediate child of the “menu” div, so it will not be yellow.

```
.menu > a { color: yellow; }
```

Finally, multiple combinators can be used. For example, a selector can specify that a certain style should be applied only to “a” tags that are children of a “li” tag that are descendents of a “menu” element:

```
.menu li > a { color: brown; }
```



Note: It is especially important to note that there is a difference between the **div.class** selector (with no space) and the **div .class** selector (with a space before the dot): the first is a single combined selector and the second is a selector with two parts and a descendent combinator. This is a common source of confusion for beginners.

Pseudo-Classes and Pseudo-Elements

Pseudo-classes and pseudo-elements are selectors of a different kind. Instead of matching elements in the element tree based on information that appears in the tree, they match elements based on other criteria.

Pseudo-classes

Pseudo-classes match elements based on certain states. For example, the a (anchor) element can have the following four states: link, visited, hover, and active. It is possible to specify a different style rule for each of these states. A pseudo-class is written as a colon (:) followed by the pseudo-class name. It is not uncommon to see styles like the following example.

```
a:link { color: blue; }
a:visited { color: purple; }
a:hover { color: red; }
a:active { color: green; }
```

- The link state matches an element whose link has never been visited.
- The visited state matches an element whose link has been visited.

- The hover state matches an element *while* the mouse hovers over it.
- The active state matches an element *while* it is being clicked.

None of these states affect the element hierarchy. Note that for the styles to be effective, they should be specified in the order in the example.

Another common pseudo-class is the “first-child” pseudo-class. This pseudo-class does match elements based on their position in the element tree, but is still considered a pseudo-class because the fact that an element is the first child of its parent is considered an arbitrary state. The first-child pseudo-class is usually combined with a tag or class selector. The following selector applies the orange color only to the first paragraph of an article.

```
.article > p:first-child { color: orange; }
```

Pseudo-elements

Pseudo-elements do not match elements; they match portions of elements. Pseudo-elements are fictitious elements that often require unique styles, but cannot be represented by using markup. For example, it is common to use small-caps for the first line of an article or book chapter. Because different screens, browsers, fonts, zoom levels, and other factors are beyond the control of the web developer or content editor, it is impossible to know how much text appears on the first line before it is actually rendered. The “first-line” pseudo-element solves that problem by explicitly instructing the browser to match the text that appears on the first line it actually renders.

The first-letter pseudo-element is similar to the first-line pseudo element. While it is technically possible to create a real element for the first-letter, it is often inconvenient and requires unnecessary effort. The first-letter pseudo-element allows designers to create fancy styles similar to those that appear in many printed books and magazines. The following rule draws only the first letter of the first paragraph of any article, red.

```
.article > p:first-child:first-letter { color: red; }
```

The before and after pseudo-elements let the CSS author add content to the page without modifying the HTML or affecting the element tree. The before and after pseudo-elements are generated elements that do not appear in the element tree. They are placed before and after the matching element in the natural flow, respectively. They are often used to add helpful icons that are not part of the page content. The before and after pseudo-elements are usually specified with the content style property. The following example adds a small image after the content of every A element that has the “external-link” class, which is a helpful indication to the user that the link references an external resource not managed by the current site.

```
a.external-link:after { content: url(/images/externallink.png); }
```

Common Styles

- The background style properties control the background's color, image, and image position
- The font and text style properties control the text's font, size, weight, and style
- The **text-align** property controls the text's horizontal alignment
- The **color** property controls the text color
- Colors in CSS can be specified in multiple ways

```
body
{
  background-color: blue;
  background-image: url('/images/background.png');
  font-family: Arial, sans-serif;
}
```

CSS defines too many style properties for us to cover all of them. This topic covers some of the most commonly used styles, especially those used to control background and text. Additional common styles are covered in the Layout topic.

Background

There are quite a few style properties for specifying the background of an element.

The background-color Property

The **background-color** property specifies the background color of an element. Its value is any valid color value, which is discussed later.

The background-image Property

The **background-image** property lets the user specify any image as the background. If the **background-image** is specified after the **background-color**, the image is visible above the color. The color is visible through any "holes" in the image, whether due to transparency or (possibly intentionally) mismatched dimensions. It is also visible before the image is loaded. The value of a **background-image** property is a url value. The following example specifies that the background is blue wherever the background.png image is not visible.

```
body
{
  background-color: blue;
  background-image: url('/images/background.png');
}
```

The background-repeat Property

By default, the dimensions of the background image are not modified based on the dimensions of the element. Instead, it is tiled across and then down to fill the background completely. Both tiling and position can be changed. The **background-repeat** style affects the tiling and can have one of the following values: "repeat", "repeat-x", "repeat-y" or "no-repeat". The default value is "repeat" which instructs the browser to tile the image, both horizontally and vertically.

The **background-position** Property

The **background-position** style specifies where to place the image initially. Its value consists of two parts: the horizontal and the vertical positions, which can each be specified by name (left and right, top and bottom, and center), absolute pixels, or percentage. The following example shows the element once (with no tiling) and sets the indent so it starts at 20 pixels to the right of the left edge of the element and close to the top.

```
body
{
    background-repeat: no-repeat;
    background-position: 20px top;
}
```

The **background** Property

Many groups of related styles in CSS can be expressed by using a *shorthand style property*. The **background** shorthand style property can specify all of these styles in one terse declaration. As with most shorthand properties, CSS is very forgiving, so neglecting to specify one or more of the constituent property values will result in default values being used in their place. IntelliSense in Visual Studio can help remember which properties are available and the correct order of the values.

```
body { background: url('/images/background.png') 20px top no-repeat; }
```

Text

HTML is primarily designed to mark up text. CSS exposes many style properties specifically for text that can provide very fine control over textual display.

The most significant properties are the various font properties. The shorthand **font** property can be used to declare them in one sweep, just as the shorthand **background** property can declare the various background properties together.

The **font-family** Property

The **font-family** property selects the general font used to draw the text. Because different browsers on different platforms and devices recognize different fonts, the **font-family** property accepts multiple values separated by commas. If the first font is unavailable, the browser tries the next, until it finds one it recognizes. If it does not find a font, it uses whatever the current font happens to be. This is a fallback system, so the text is always rendered even if no specified font is found. Any number of font families can be specified as fallbacks.

Every browser is equipped with generic font families, which are general mutually exclusive categories that every font family can be categorized by. The most common generic families are "serif", "sans-serif" and "monospace". Serif fonts are relatively ornate fonts that have embellishments at the tips of the letters, such as Times New Roman, the archetypal serif font. Arial is a sans-serif font, a font with simple line endings and no embellishments. Monospace fonts are fonts in which every character has the same width and height. The "cursive" and "fantasy" generic fonts are rarely used.

Every **font-family** declaration should be suffixed with a generic family so the browser can use a default font that belongs to the same family, if the requested font family is unavailable. The following declaration specifies that the page should use the Arial font or whatever sans-serif font it has, if Arial is missing.

```
body { font-family: Arial, sans-serif; }
```

The font-size Property

The **font-size** property can specify an absolute value, such as "36px" or "30pt" or "large", or a relative value, such as "150%" or "larger" or "1.5em". Relative sizes are relative to the size of the containing parent element. Units are discussed in the next topic.

The font-style Property

The **font-style** property specifies whether the font is "normal", "italic" or "oblique" (another italicized variant).

The font-weight Property

The **font-weight** property specifies the thickness or darkness of the text. It can accept absolute values, such as "bold", or a relative value, such as "bolder". The absolute value can also be a number between 100 and 900 (only round numbers divisible by 100).

The text-decoration Property

The **text-decoration** property specifies whether to draw a line under, over, or through the text. It accepts the values "none" (no line should be drawn), "underline", "overline" and "line-through" (effectively strikethrough). We strongly recommended not to use the additional "blink" value specified in the standard, because it is not supported by all browsers and generally considered a bad practice. Anchors ("a" tag) use the "text-decoration: underline" declaration by default.

The text-align Property

The **text-align** property defines the horizontal alignment of the text. It is only effective in block elements and cannot affect the vertical alignment. Its value can be "left", "right", "center" or "justify".

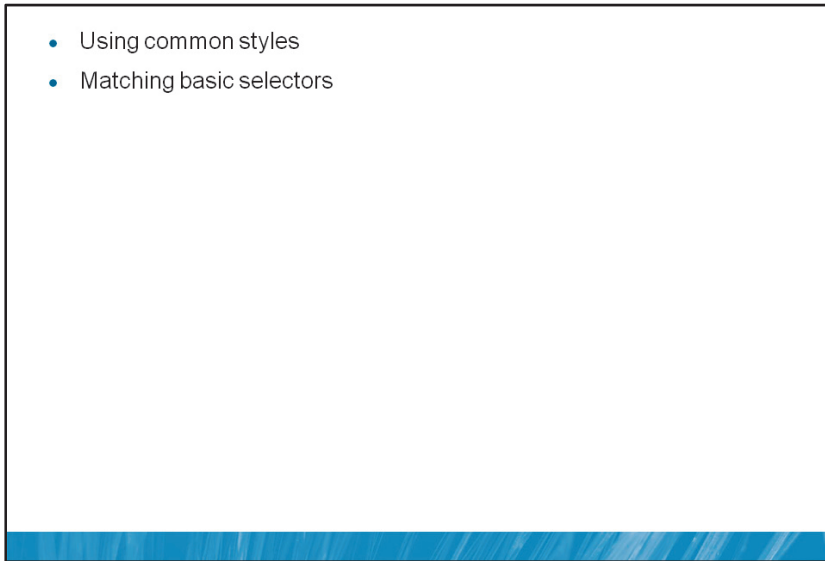
Colors

The **color** property specifies the foreground color of the text. To draw a highlight, use the **background-color** property. There are a number of ways to specify colors in CSS:

- Use the name of a known color: red.
- Use a 3-digit RGB value: #f00.
- Use a more precise 6-digit RGB value: #ff0000.
- Use the rgb function with absolute values: rgb(255, 0, 0).
- Use the rgb function with percent value: rgb(100%, 0%, 0%).

All of these examples represent the same color. CSS provides another option based on "system colors" which is inconsistent across browser implementations and should not be used. CSS3 adds additional options, which are covered in a later module.

Demonstration: Common Styles



This demonstration shows how to:

- Apply a background color to an element.
- Apply a font and color to text.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file, add the following HTML content to the **body** element.

```
<h1>The Header</h1>
<div id="content">The Content</div>
<div class="footer">The Footer</div>
```

4. In the HTML file, add a **style** element inside the **head** element before the **head** element's closing tag.

```
<style type="text/css">
  h1
  {
    font-family: Arial sans-serif;
    font-size: 36px;
    text-decoration: underline;
    color: brown;
    text-align: center;
  }
  #content
  {
    background-color: brown;
    color: white;
    font-weight: bold;
    height: 200px;
  }
  .footer
  {
```



```
    font-size: 0.6em;  
    color: gray;  
    text-align: right;  
  }  
</style>
```

5. Save the HTML file.
6. Run the saved file.

Lengths and Units

- CSS provides fine control over element sizes and lengths
- Pixels, denoted by “px”, are the most common unit of measure
- Points, denoted by “pt”, and other absolute units, are preferable in print
- Percentages and other relative units are preferable on screen
- Ems are relative units based on font size and better for platform compatibility

Overview

When a browser lays out an HTML document, it primarily tries to let the content “flow” naturally. Text and images, for example, fill up the available space and the browser breaks lines as necessary. When the browser window is resized, the content is repositioned so it continues to flow. HTML layout is designed with the assumption that the dimensions of the output medium (the screen or printer) are not necessarily known ahead of time.

The main repercussion of this layout model is that precise control over the lengths and dimensions of various elements is very difficult. Web designers and developers have to account for the great variations between, for example, large desktop monitors and tiny mobile screens and printers of various sizes. It can be quite a challenge to style a webpage so it looks good in all these different form factors.

CSS allows developers to specify lengths in absolute and relative terms, using many different measurement units. These lengths can be applied to almost every element, including text and images, block and inline elements, and the body and pseudo-elements.

Every number in CSS that denotes a size or length must have a unit designation. Unit designations are affixed to the number values without a space. So 10 pixels is denoted as “10px”. Note that many sites use a mixture of units, including pixels, ems, percentages, and others. Some units denote absolute sizes and others denote relative sizes.



Best Practice: The number 0 is an exception and does not require a unit. However, it is a best practice to specify units for 0 anyway.

Pixels

The most common unit of measure is the pixel, an absolute size denoted by the “px” suffix. The actual size of a pixel depends on a number of factors, including the device’s screen resolution and the zoom level in the browser. Pixel sizes vary between devices, monitors, operating systems, and browsers. Despite these

differences, pixels are still considered the most viable option for specifying sizes, mainly because they produce relatively predictable results.

Points and other absolute units

Points, denoted by the “pt” suffix, are not used as often as pixels. They are usually used for text because fonts in operating systems, word processing applications, and print use points as their base unit of measure. Fonts have also traditionally tended to look better when their size is specified in points because they do not need to scale by fractional values. HTML documents often use points to specify font sizes.

Points in CSS have a very specific size: they are defined as being equal to one seventy-second of an inch. CSS also defines additional absolute sizes, including millimeters (mm), centimeters (cm), inches (in), and picas (pc), which can all be converted to points very easily. These units are all available in CSS, but their use is discouraged, except in very specific cases.

Though points and other absolute units produce very accurate results when printed, they are not as accurate on screen because of the great variation in monitor resolutions and operating systems. Browser implementations for pixels vary, and they do not scale well for screen readers or mobile browsers.

Percentages

Percentages, denoted by the percent symbol (%), are relative sizes. Whenever a percentage is specified, the browser calculates the actual size based on the size of its parent. Judicious use of absolute and relative sizes helps produce layouts that are easier to resize as needed and take accessibility into consideration.

Ems

Ems, denoted by the “em” suffix, are another relative size unit. Unlike percentages that are calculated relative to the size of the parent, ems are calculated based on the current font size. They are specified multiples of the current font size. For a font size of 12 points, for example, **1em = 12pt**, **2em = 24pt**, and **0.5em = 6pt**.

Ems are extremely useful on websites that present a lot of text, because they make the layout more predictable. For example, if the width of a paragraph containing text is specified as 30%, the amount of text that fits on one line changes from browser to browser and monitor to monitor. However, if it is specified as 20em, every browser on every monitor displays the same number of characters on one line.

Layout

- The CSS Box Model defines a number of boxes that surround the content:
 - Content
 - Padding
 - Borders
 - Margins
 - Offset
- The display property determines how an element is rendered
- The visibility property determines whether an element is shown
- The float and clear properties let elements be positioned outside the normal flow
- Vertical alignment can be implemented by using various techniques
- Overflow properties control scrollbars and whether overflow content is shown

The CSS Box Model

Content

When the browser renders HTML elements, it uses the concept of a box to lay it out. Each element is rendered as though it is a rectangular box with multiple outer layers, like a gift box wrapped with several layers of wrapping paper. Whatever is written inside the HTML element is the content of the element and is drawn in the inner-most box. The content box is surrounded by a few layers, each of which has a specific role in the box model.

The size of the inner content area ultimately defines the size of the rendered element. The browser tries to render the element in the available space if no size styles are specified. CSS provides a number of styles that can be used to define the size of an element content. The style properties that determine the size of the content itself are **width** and **height**. Any size unit can be specified for them. In the following example, the browser renders the size of any element with the “footer” class name at 80 percent of the width of the element’s parent’s content width, and as high as 4 characters (in whatever their actual size happens to be), stacked one on top of the other.

```
.footer
{
  width: 80%;
  height: 4em;
}
```

The browser may not always be able to render the content at the requested size. It provides a few style properties that can help determine the limits for the browser. The min-width, max-width, min-height, and max-height properties provide the browser with strict limits. They can be used to lay out the content in a flexible manner that stretches gracefully as the browser window expands and shrinks, without ruining the visual appeal of the site. The following example renders the “main-content” area of the page at 80 percent of the available width, but specifies the width should never be less than 300 pixels wide so it cannot be hidden or made ugly by a too-small browser window. Note that only a min-height property is specified

for the height. This is a useful technique to ensure that a content area is visible even when there is little or no content.

```
.main-content
{
  width: 80%;
  min-width: 300px;
  min-height: 400px;
}
```

If min-width or min-height are specified and the browser determines that it does not have enough available space, it renders the element at the width or height specified by the respective property. It also shows horizontal or vertical scrollbars, respectively, on the browser window to let the user scroll to see the hidden content.

Padding

The next layer outside the content area is the padding. Padding specifies the amount of space between the edge of the content area and the element's border. The padding area shares the background of the content area, so the following CSS specifies that the content should have a blue background and the background color should extend 20 pixels in all directions beyond the actual width and height of the content.

```
.main-content
{
  background-color: blue;
  padding: 20px;
}
```

The padding property is a shorthand property that abbreviates the padding-top, padding-right, padding-bottom, and padding-left properties. When the shorthand property is used, CSS fills the property values according to a very specific set of rules, in longhand.

When only one measurement is specified, all sides receive the same value, so **padding: 20px;** is equivalent to the following example.

```
.main-content
{
  padding-top: 20px;
  padding-right: 20px;
  padding-bottom: 20px;
  padding-left: 20px;
}
```

Note that the sides are always specified in the same order: top, right, bottom, and left. This is a common convention in the CSS standard for all properties that refer to the sides of an element.

When two measurements are specified, as in **padding: 20px 40px;**, the first measurement refers to the top and bottom, and the second measurement refers to the right and left. This is equivalent to the following example.

```
.main-content
{
  padding-top: 20px;
  padding-right: 40px;
  padding-bottom: 20px;
  padding-left: 40px;
}
```

When three measurements are specified, as in **padding: 10px 20px 30px;**, the first size always refers to the top and the third always refers to the bottom. The second size refers to both the right and the left. This is equivalent to the following example.

```
.main-content
{
  padding-top: 10px;
  padding-right: 20px;
  padding-bottom: 30px;
  padding-left: 20px;
}
```

Finally, when all four values are specified, they are assigned to the sides in the normal order, so **padding: 10px 20px 30px 40px;** refers to the top, right, bottom, and left, in that order as shown in the following example.

```
.main-content
{
  padding-top: 10px;
  padding-right: 20px;
  padding-bottom: 30px;
  padding-left: 40px;
}
```

Note that the same units are used for convenience. There is no such requirement. Each of the sides can use any unit. It is legal, for example, to specify **padding: 10px 20pt 30em 40%.**

Borders

Outside the padding is the border area. The border is specified by using a number of properties. Each side has a border and each border has a width, a style, and a color. The width is any ordinary size unit. The color is any color unit. The style is one of the special border style values. The most common border styles are "**none**", which specifies that there should be no border at all. This is equivalent to setting the border width to 0, "**hidden**", which specifies that the border is present, but invisible and "**solid**", which determines that the border is visible as a single line.

Each side has its own set of properties that specify each of these values. For example, **border-top-width**, **border-top-style**, and **border-top-color** specify all the properties of the top border. The shorthand property, **border-top**, can be used to specify all three at once. As usual, omitted values from the shorthand property are filled in with default values. There are similar properties and shorthand properties for the other sides also..

```
.main-content
{
  border-top: solid 1px red;
  border-right: inset 2em #0f0;
  border-bottom: double 3% #00ccff;
  border-left: groove 4pt; /* the color is filled in with the default */
}
```

Instead of specifying each side individually, the shorthand properties **border-width**, **border-style**, and **border-color** can be used to specify all the widths, styles, and colors, respectively of the entire surrounding border. Each of these properties can specify one, two, three, or four values, which are applied to the borders in the same order by using the same logic as the padding that is applied to each side.

The last shorthand property is **border**. The **border** property can be used to specify the same width, style, and color for all the borders and it is specified the same way that **border-top** is specified. The following CSS specifies that the entire element is surrounded by a single solid green border 2 pixels wide.

```
.main-content
{
  border: solid 2px green;
}
```

Margins

The last layer is the margin, which is outside the border. The margin determines the amount of space between elements. Unlike the padding, which shares the background with the content area, the margin shares the background with the content area *of the parent element*. The margin is specified exactly the same way that the padding is specified, and it has a longhand size property for each side and a shorthand **margin** property.

Offset

The area outside the margin is not really part of the box model, but it is very useful for moving the element relative to its position in the general flow. The offset property is very similar to the padding and margin properties. The following CSS moves the element a bit to the left.

```
.content-area
{
  offset-left: -5px;
}
```

CSS contains quite a few properties that are specifically designed to affect the layout of the elements on the page. Each layout property affects the layout in a different way. This topic introduces the most common and useful layout properties.

The display property

The **display** property instructs the browser which layout algorithm to use to lay out a specific element. Every element has a default layout mechanism. Most elements are either inline elements or block elements. The **display** property can be used to change the layout algorithm used to render an element. The most common display values are "**block**" and "**inline**". The block display value specifies that an element should be rendered as a block element, which means that the element should start a new line, should take up as much space as it can horizontally, and that new content should appear beneath it. The inline display value specifies that an element should be rendered in the normal flow, next to the previous element, as though it is part of the text.

The **div** and **span** elements are the archetypal structural elements used for controlling block and inline flow in a document. The **div** element specifies, by default, that the display is "block" and the span specifies "inline". Using the display style property, these roles can be reversed. The following example is given to illustrate the switch; using the style in this way is not recommended.

```
div
{
  display: inline;
}
span
{
  display: block;
}
```

Another useful display value is the "inline-block" value, which instructs the browser to lay out the content inside the element as though the element is a block element, but to position the element in the document flow as though it is an inline element.

When display is set to "none", it instructs the browser not to render the element at all. The element is completely hidden and the adjacent elements are rendered as though the element is not specified in the document at all. This is useful for creating content that is shown and hidden, dynamically.

The visibility property

The visibility style determines whether an element should be visible and has three possible values: visible, hidden, and collapsed. When visibility is set to visible, it is rendered as it normally would be. When it is set to collapsed, it is not rendered at all, as though display is set to none. Finally, when the visibility is set to hidden, the element is rendered and takes up as much space as it needs, except that it is not visible. It affects the layout, but cannot be seen.

The float and clear properties

The float and clear properties complement each other. They each accept the values "left", "right" and "none". The clear property also accepts "both". The float property instructs the browser to ignore the normal position of the element in the document flow and to place it on the left or right side of the content. To place a sidebar on the right side of the page, the following CSS might be used.

```
div.sidebar
{
  float: right;
}
```

When float is set to none, it instructs the browser to leave the element in the natural flow.

The clear property has the opposite behavior. It instructs the browser not to allow floats on the specified side, or on both sides. When clear is set to "none", floats can be placed on either side of the element. When it is set to left, floats *cannot* be positioned to the left of the content. Floats appear either above or beneath the element in these cases.

Vertical Alignment

Text is notoriously difficult to align vertically by using CSS. However, text in table cells generally aligns vertically very well. In a table cell, the vertical-align property can be set to "top", "bottom", "middle" or "baseline". Baseline is an imaginary line that is used to lay out text with different font sizes at a common vertical position.

One technique to use to vertically align text that is not in a table cell is to set the element's display property to "table-cell". The following example aligns text in every div to the bottom. Note that unrestricted use of this technique is not recommended because the use of the "table-cell" property value has other consequences.

```
div
{
  display: table-cell;
  vertical-align: bottom;
}
```

Another technique used to align text vertically is the use of the line-height property. The line-height property specifies the height of the line used to draw text. Usually, the line height is directly related to font size, but when the line height is greater than the font size, the text is automatically laid out vertically in the middle.

Overflow

When content is too large for the content box, the shorthand **overflow** and longhand **overflow-x** and **overflow-y** properties can be used to control what happens to the overflowing content. The overflow properties can specify that the overflowing content should be “visible” (the default value) or completely “hidden” from view. They can also specify that a “scroll” bar should always be shown at the edges of the content area to let the user scroll when content overflows. The last value, “auto”, instructs the browser to only show the scroll bars if content really overflows. When the shorthand **overflow** property is used, it implicitly specifies both longhand properties. The **overflow** property is often used together with the **max-width** and **max-height** properties that specify the maximum size of the content area. The following example constrains the size of the content area and adds a vertical scrollbar.

```
.content
{
  overflow-x: hidden;
  overflow-y: scroll;
  max-width: 350px;
  max-height: 200px;
}
```

Lesson 4

Introduction to JavaScript

- What Is JavaScript?
- Primitive Types
- Variables
- Functions
- Objects
- Scope
- The Document Object Model (DOM)
- Working with JavaScript
- Ajax and JSON

JavaScript is the programming language of the World Wide Web. It is the only programming language supported by all modern desktop and mobile browsers. Just as HTML describes the content of the page and CSS defines its style and layout, JavaScript implements its behavior. This lesson introduces JavaScript and teaches students how to create interactive webpages, in accordance with current best practices.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the basic JavaScript principles used in the course.
- Explain JavaScript primitive types.
- Describe JavaScript variables.
- Describe JavaScript functions.
- Explain JavaScript objects and how to create them.
- Explain JavaScript scope.
- Describe the ways to add JavaScript to a page.
- Describe the Document Object Model (DOM) and how it relates to JavaScript.
- Explain Ajax and JSON.

What Is JavaScript?

- Stepping Outside the Comfort Zone
- Dynamic, Scripting, Object-Oriented, Functional Language
- Browser Integration and the DOM

Stepping Outside the Comfort Zone

JavaScript has a unique combination of characteristics that programmers should understand so that they can use it effectively. Programmers whose main experience is with static compiled programming languages, such as C# and VB.NET, should be aware that JavaScript represents a significant departure from the kind of languages they are familiar with. There is often a psychological barrier for experienced programmers who encounter languages with any of the characteristics that JavaScript has. This may require programmers to step outside their comfort zone. They should be encouraged to use an open mind as they approach JavaScript and to learn it as best as they can before they pass judgment on it. Arguing over the merits of the different programming paradigms before they are well-understood is detrimental to the educational experience and should be avoided.

Language Characteristics

JavaScript is a dynamic language. The program creates and modifies its own types and structures at runtime. This feature makes many otherwise cumbersome tasks trivial and has made JavaScript very easy to use.

JavaScript is a scripting language. The browser parses and runs the JavaScript code in the script files referenced by the HTML document and they pass instructions to the browser.

JavaScript is an object-oriented language. Objects in JavaScript can have properties and methods, and they can interact with each other. Because the language is dynamic, object structures are created and modified dynamically, just like every other structure in JavaScript. However, JavaScript is considered a classless system, so objects do not derive from or extend classes. Instead, it has a different inheritance system called prototypal inheritance, which is covered later in the lesson. It is important to note that objects are optional in JavaScript, which makes it easier for non-professional programmers to build simple websites. Programmers are strongly encouraged to use objects as much as possible to write more robust and maintainable code. The proper use of objects is crucial to building modern, complex, and interactive websites.

JavaScript is a functional language. Functions are very powerful constructs in JavaScript. They can be manipulated just like variables and objects. They can be stored in variables and properties, passed around

freely, and be constructed dynamically and run anonymously. Unlike many functional languages, though, JavaScript functions produce side-effects, are not immutable, and lack support for pattern matching and other features commonly associated with functional programming languages. The relationship between functions and objects in JavaScript is also unique, so understanding how functions work in JavaScript is vital to working with both.

Browser Integration

Anybody can read and modify the program while it runs in the browser. All major browsers have tools and extensions, such as the bundled Developer Tools in Internet Explorer. The tools and extensions make it exceptionally easy to debug JavaScript code inside the browser. The Developer Tools have interactive debuggers that support placing breakpoints on JavaScript code inside the browser, tracking variable values with watches, and viewing the stack trace. The Developer Tools all have a Console feature, which lets the user enter JavaScript code and commands directly into the browser. The context in which code in the console is run is the same context used to run all the code on the page. One useful debugging technique is to add log statements to the code to track select operations. These log statements appear on the console.

```
console.log("This statement appears on the JavaScript console.");
```

This feature is invaluable not only for debugging errant code, but also for learning about the language. This lesson demonstrates many concepts with code examples that can be entered directly into the console.

The browser hosts the JavaScript code and manages its execution and interface with the HTML document. The browser's JavaScript engine parses, interprets (or JIT compiles), and runs the JavaScript. It is also protective and limits JavaScript access to external services, including the file system, to ensure malicious programs cannot damage the end-user's computer. JavaScript code only has access to the APIs exposed by the browser.

Every modern browser creates an internal in-memory representation of the HTML document and exposes that model as a hierarchical element tree called, the Document Object Model, or DOM. The DOM is designed to be read and manipulated by JavaScript code. By manipulating the DOM, a JavaScript programmer can modify almost every aspect of a webpage, and even rewrite it completely on the fly. The DOM is the interface that enables programmers to turn static HTML into a vibrant, dynamic, and interactive site. JavaScript and the browser have many features whose goals are to facilitate interaction with the DOM.

Primitive Types

- Five Primitive Types
- Type Conversions
 - Weakly-Typed Language
 - Type Coercion
 - Strong Equality

Five Primitive Types

JavaScript has five primitive types: boolean, number, string, null, and undefined. The primitive types are all immutable and are all passed by value. The first three (boolean, number, and string) also have corresponding built-in objects called Boolean, Number, and String, respectively. It is important to note that the primitives and their corresponding objects are designed to work together, but do not behave the same. The built-in objects are introduced in a later topic.

The boolean type

There are two Boolean values, **true** and **false**. Conditional statements, such as **if** and **while**, evaluate Boolean values. Comparison operators, such as greater than (>), less than (<), equal to (==) and strictly equal to (===) all return Boolean values. The following are legal Boolean expressions.

```
true;  
false;  
true == true;  
1 > 0;  
42 == "42"; // equal to -> returns true  
42 === "42"; // strictly equal to -> returns false
```

The number type

All numbers in JavaScript are represented by the number type. Internally, they are all 64-bit floating point numbers. This includes whole numbers, decimal numbers, signed numbers, and unsigned numbers.

JavaScript supports several notations for numbers:

- Decimal (base 10) numbers are written normally (for example, "4" or "4.5").
- Octal (base 8) numbers are written with a "0" prefix (for example, "010").
- Hexadecimal numbers (base 16) are written with a "0x" prefix (for example, "0x1F").
- Scientific notation is also supported (for example, "4e+3").

JavaScript also has a few special named numbers:

- Infinity—Infinity is a numeric value in JavaScript. It can be tested for positive or negative infinity by using the Math object. Dividing by zero also produces the value, Infinity.
- Not a Number (NaN)—NaN is a numeric value in JavaScript. It is the value returned when converting non-numeric strings and other values to numbers. It is also given as the result of certain calculations.

JavaScript supports common mathematical operators on numbers, including addition (+), subtraction (-), multiplication (*), division (/), and modulus (%). It honors common mathematical operator precedence, including parentheses. Many additional numeric operations can be performed with the Math object, which is covered later.

The following expressions all represent legal numeric expressions in JavaScript.

```
0;
1;
1.5;
+1.5;
-3.777777;
4 + -3.2;
9 % 2;
4e+3;
4e-3;
Infinity;
NaN; // represents a "Not a Number" value
1 / 0; // returns Infinity
Infinity; / Infinity // return NaN
010; // octal: returns the number 8
0x10; // hex: returns the number 16
2 + 3 * 4; // returns 14
(2 + 3) * 4; // returns 20
9 % 2; // returns 1
```

The string type

JavaScript string values represent Unicode text. String literals can be denoted by either the double quotation mark ("...") or the single quotation mark (apostrophes ('...')), so long as the same character is used to open and close the string literal. There is no character type in JavaScript; a character is simply a string with one element.

The escape character is the backslash (\). To use a double quotation or single quotation character in a string, escape it as follows: \" or \', respectively.

Strings can be concatenated by using the plus (+) operator. The String object, which is covered later, provides many additional string manipulation methods.

The following expressions all represent legal string expressions in JavaScript.

```
""; // double quotes - the empty string
''; // single quotes - also the empty string
"Hello, world!";
"This string \n has an escaped new line character";
"The secret is " + "42"; // simple concatenation
```

The null type

The null value in JavaScript is not the default value of unassigned variables and properties, as it would be in most languages. That purpose is served by the undefined value. The null value is a value the programmer uses to explicitly denote an object with no value. It is inapplicable to non-object types.

The undefined type

The undefined value is the default value given to variables and other containers that have not been assigned a value. This applies to variables, function arguments, and object properties and methods. The undefined value in JavaScript is very similar in meaning and behavior to the null value in other languages. Usage of the undefined value is demonstrated in the next topic.

Variables

- Declaring Variables
- The **typeof** operator
- Assigning Variable Values

```
var enigma;
typeof enigma; // returns "undefined"

enigma = true;
typeof enigma; // returns "boolean"

enigma = 42;
typeof enigma; // returns "number"

enigma = "mystery";
typeof enigma; // returns "string"
```

Declaring variables is very simple. Variables are declared by using the "var" keyword.

```
var enigma;
```

When a variable is declared without being assigned a value, its type is **undefined**. To check if a variable value is undefined, use the **typeof** operator.

```
typeof enigma; // returns "undefined"
```

The **typeof** operator can be used to check the type of any value.

```
typeof true; // returns "boolean"
typeof 42; // returns "number"
typeof "mystery"; // returns "string"
```

However, its results are inconsistent, or at the very least, unhelpful, after objects are involved, as is explained when they are introduced. An alternative method for checking types is also shown. Note that the **typeof** operator is the only way to check if a variable has not been declared at all, because every other method (including the alternative method mentioned above) causes JavaScript to raise a **ReferenceError**.

The following example demonstrates the difference between using the **typeof** operator to check an undeclared variable and trying to reference an undeclared variable.

Using and Checking Undeclared Variables

```
if(typeof undeclared == "undefined") {
  console.log("This variable has no value or has not been declared!");
}
isUndefined(undeclared); // this raises a ReferenceError even if the isUndefined function exists
```

To assign a value, use the assignment (**=**) operator.


```
var enigma; // declaration
enigma = "mystery"; // assignment
```

The value can also be assigned while declaring the variable.

```
var enigma = "mystery"; // declaration and assignment
```

After the value is assigned, it is stored inside the variable. In JavaScript, only values have types; variables do not. When a variable is checked for its type, it actually returns the type of its value. Because JavaScript is a dynamic language, a single variable can be assigned a value of any type and then reassigned with another value of any type.

The following example shows how the type of a variable changes when its value changes.

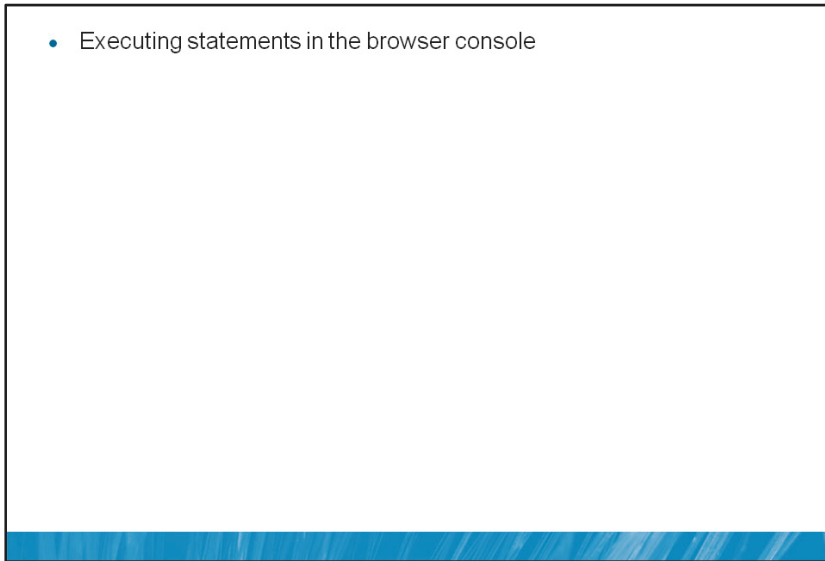
Dynamically Changing a Variable Type

```
var enigma;
typeof enigma; // returns "undefined"
enigma = true;
typeof enigma; // returns "boolean"
enigma = 42;
typeof enigma; // returns "number"
enigma = "mystery";
typeof enigma; // returns "string"
```



Note: It is technically possible in JavaScript to declare a variable without using the **var** keyword. However, this is not equivalent to a proper declaration and does not have the same behavior. It usually has unintended consequences, as is demonstrated in the topic on Scope, and very often introduces difficult bugs. Omitting the **var** keyword is a *very* unacceptable practice and should always be considered an error. Never omit the **var** keyword when declaring variables.

Demonstration: Using the Console



This demonstration shows how to:

- Use the Internet Explorer Developer Tools console.
- Run statements inside the browser directly.

Demonstration Steps

1. Open Internet Explorer.
2. Press F12 to open the Internet Explorer Developer Tools.
3. If the developer tools opens in a separate window, instead of in a pane at the lowermost part of the browser window, click the Pin button at the upper-right part of the window.
4. Click the **Console** tab.
5. Next to the >> symbol, enter the following code and then press **Enter** to show an alert message.

```
alert("Hello");
```

6. Observe the popup and then click OK to close it.
7. Enter the following line of code to print the type string of a number.

```
typeof 42;
```

8. Observe the result of the expression ("number") printed in the console.
9. Enter the following line to declare and assign a new variable called **message**.

```
var message = "Hello";
```

10. Enter the following line to display the value of the message variable.

```
message;
```

11. Observe the value of the variable displayed in the console.
12. Enter the following line to show the value of the variable in an alert.

```
alert(message);
```

13. Click **OK** to close the alert.

Functions

- Declaring Functions
- Passing and Returning Values
- Passing Functions Around
- Anonymous Functions
- Nested Functions

```
// the function declaration
function doSomething() {
    // code goes here
}

doSomething(); // calling the doSomething function
```

Functions are the most basic unit of re-usable code in JavaScript. When a function is called, the statements in the function are executed. Simple JavaScript functions are very similar to functions in other languages, while advanced JavaScript functions have some unique characteristics that stand out among them.

Declaring Functions

The most basic JavaScript function is declared by using the **function** keyword, followed by the function name, parentheses, and braces. Whatever code is placed between the braces (inside the function) is executed when the function is called. Functions are called by writing their name, followed by parentheses.

```
// the function declaration
function doSomething() {
    // code goes here
}

doSomething(); // calling the doSomething function
```

The function provides scope, so variables that are declared locally inside the function are only available inside the function. Although this point may seem trivial, it is actually quite important to understanding some of the function's more advanced functionality. This point will be returned to shortly.

```
function doSomething() {
    var data = "local variable";
    console.log(data); // this is a legal expression - data is "in scope"
}

doSomething();
console.log(data); // data is undefined in this scope, so this statement raises a
ReferenceError
```

Passing and Returning Values

Function Arguments

Values can be passed to functions as arguments. Function parameters ("formal arguments") can be declared as a comma-separated list of names between the parentheses. The arguments behave just like

local variables declared inside the function, though the **var** keyword is not used for their declaration. The following function has two parameters called, **message** and **value**. Note that just like with variables, their values can have any type. Arguments are passed to the function by placing them in the parentheses when calling the function.

```
function logArguments(message, value) {
  console.log(message); // prints "The secret is"
  console.log(value); // prints 42
}
logArguments("The secret is", 42);
```

The arguments keyword

Arguments in JavaScript are rather flexible. The number of arguments passed to the function is not required to match the number of parameters declared on the function. It is legal to pass *fewer* arguments than were declared, in which case, arguments that received no value are **undefined**. It is also legal to pass *more* arguments than are declared.

All the arguments passed to the function can be accessed through the **arguments** keyword, which is available only inside functions. The **arguments** keyword represents a special object that behaves similar to an array (but is not an array), and it contains a list of all the arguments passed to the function, whether declared or not. One way to think about the **arguments** object is as an implicit parameters object that automatically includes arguments passed to declared parameters. To determine the number of actual arguments passed to the function, check the value of the **arguments.length** property.

```
function print(first, second) {
  console.log(first);
  console.log(second);
  console.log(arguments.length);
  console.log(arguments[0]); // refers to the "first" parameter
  console.log(arguments[1]); // refers to the "second" parameter
  console.log(arguments[2]); // has no corresponding parameter
}
print(1); // the following values are printed: 1, "undefined", 1,1, "undefined",
"undefined"
print(1, 2); // the following values are printed: 1, 2, 2, 1, 2, "undefined"
print(1, 2, 3); // the following values are printed: 1, 2, 3, 1, 2, 3
```

The items in the **arguments** object are interchangeable with their matching parameters. Modifying one, modifies the other.

```
function print(value) {
  console.log(value); // prints 1
  console.log(arguments[0]); // prints 1
  value = 2; // change the local variable value
  console.log(value); // prints 2
  console.log(arguments[0]); // prints 2
  arguments[0] = 3; // change the value in the arguments object
  console.log(value); // prints 3
  console.log(arguments[0]); // prints 3
}
print(1);
```

Note that the **arguments** keyword is not a reserved word in JavaScript. It can be overridden by declaring a local variable or a parameter with the same name. This is unacceptable practice and should be avoided.

```
function printOverwrittenArguments(arguments) {
  console.log(arguments); // prints 1
```

```

    console.log(arguments[0]); // prints "undefined" instead of 1
  }
  function printBuiltInArguments(value) {
    var arguments = 2;
    console.log(value); // prints 1
    console.log(arguments[0]); // prints "undefined"
  }

  printOverwrittenArguments(1);
  printBuiltInArguments(1);

```

The return keyword

To return a value from the function, use the **return** keyword inside the function. A value of any type can be returned. A function can contain multiple **return** statements, although the first **return** reached ends the function execution. If no return statement is reached before the function ends, an implicit value of **undefined** is returned. There is no such thing as a void function in JavaScript, although the return value can always be ignored.

```

function isTrue(number) {
  if(number === 1) {
    return true;
  }
  if(number === 2) {
    return false;
  }
  // implicit return undefined
}
console.log(isTrue(1)); // prints true
console.log(isTrue(2)); // prints false
console.log(isTrue(3)); // prints "undefined"

```

Passing Functions Around

Functions are Values

Functions in JavaScript are values, like any of the primitive types, and can be stored in variables that can be passed around. To assign a function to a variable, simply place its name, without the parentheses, to the right of the assignment operator. To call the function by using the variable, simply append parentheses to the variable and pass any arguments, as you would for a normal function call.

```

function doSomething(data) {
}
var functionReference = doSomething; // this stores a reference to the "doSomething"
function in the "functionReference" variable
functionReference(1); // call the "doSomething" function

```

Functions can also be passed to other functions.

```

function callFunction(functionReference) {
  functionReference(); // call the "doSomething" function
}
function doSomething() {
}
callFunction(doSomething); // pass the "doSomething" function to the "callFunction"
function

```

Anonymous Functions

Functions can be created on the fly and assigned to a variable, without being declared. Functions created in this manner are unnamed and are called *anonymous functions*. In the following example, an anonymous function is created and stored in the variable, `print`. The function is called the same way any function would be called.

```
var print = function(message) {
  console.log(message); // prints "Hello"
}
print("Hello");
```

Functions can also be passed anonymously to other functions, and even to other anonymous functions.

```
var callReferencedFunction = function(functionReference) {
  console.log(functionReference()); // prints "World", the result of calling the
  // anonymous function stored in "functionReference"
}
// call the anonymous function stored in the "callReferencedFunction" variable and
// pass it another anonymous function as an argument
callReferencedFunction(function() {
  return "World";
});
```

The function type

To test whether a variable or argument contains a function, the **typeof** operator can be used. Function values have the type, "function". As noted earlier, the **typeof** operator can produce inconsistent results when working with objects, so its use is discouraged.

```
function declaredFunction() {
}
var functionVariable = declaredFunction;
var anonymousFunctionVariable = function() {
}
console.log(typeof declaredFunction); // prints "function"
console.log(typeof functionVariable); // prints "function"
console.log(typeof anonymousFunctionVariable); // prints "function"
```

Nested Functions

Functions in JavaScript can be nested, that is, a function can contain other functions, which are only available (in scope) inside that function. They are not available outside the parent function, although nested functions can see other nested functions in the same scope. Nested functions can also access variables and arguments from the parent function.

```
function parent() {
  console.log("inside parent");
  var parentData = 1;
  function nested() {
    console.log("inside nested");
    console.log("parentData is " + parentData);
  }
  nested(); // call the nested function
}

parent(); // this call prints "inside parent", followed by "inside nested" and then
"parentData is 1"
nested(); // raises a ReferenceError, since there is no nested function in the
current scope
```

Because nested functions cannot be seen from outside the parent, they very often are used to encapsulate behavior. However, it is important to remember that functions, whether they are nested or not, are values in JavaScript, so they are created and destroyed as necessary. Nested functions are only created when their parent function is called. This means if an outer function is called many times, the nested functions are also created and destroyed that many times. Repeatedly creating and destroying these functions affects performance.

While there are justified uses for nested functions, and very popular frameworks like jQuery make use of them, it is a best practice to avoid them, if at all possible. There are a number of ways to achieve similar behavior without affecting performance. The simplest is to move the nested functions out of the parent function. The nested functions are no longer hidden. Also, they will be able to see the local variables and arguments of the parent function, so they have to be passed directly. The advantage is that only one instance of the nested function is created, which significantly improves performance.

```
function nested(data) {  
    console.log("inside nested");  
    console.log("data is " + data);  
}  
function parent() {  
    console.log("inside parent");  
    var parentData = 1;  
    nested(parentData);  
}  
parent(); // this prints "inside parent", followed by "inside nested" and then "data  
is 1"  
nested(); // this prints "data is undefined"
```

A better technique to encapsulate behavior is to use objects, which are introduced in the next topic.

Objects

- Objects have properties and methods
- Use the **new** keyword or *literal object notation* to create an object
- Use the **this** keyword to refer to object members
- The global object is the browser window
- Functions are objects
- Calling a function with the **new** keyword converts it into a constructor
- Objects inherit from prototypes, not classes
- The **instanceof** and **typeof** keywords perform type checks
- The built-in **String**, **Math**, **Date**, and **Array** objects perform common services

```
var car = new Object();
car.speed = 0; // the speed property is created and initialized
car.drive = function () { // the drive method is assigned an anonymous function
    this.speed = 60; // from 0 to 60 in a few nano-seconds isn't bad
};
console.log(car.speed); // prints 0
car.drive(); // execute the drive method
console.log(car.speed); // prints 60
```

What Is an Object?

An object in JavaScript is a dynamic bag of variables and functions, called properties and methods, respectively. Properties and methods can be added to a single object. An object also inherits properties and methods from its prototype. JavaScript does not have classes; prototypes are JavaScript's mechanism for inheritance.

Creating Objects

To create an object in JavaScript, use the **new** keyword with the **Object** constructor.

```
new Object();
```

To store the object in a variable, simply assign it as you would assign any value. Objects in JavaScript are reference types, so the variable actually stores a reference to the object, and not the object itself. Also, two variables can hold a reference to the same object.

```
var first = new Object();
var second = first;
first === second; // returns true
```

Note that each call to `new Object()` creates a completely new object, which is not equal to any other object.

```
new Object() === new Object(); // returns false;
```

Objects are more useful with properties and methods, which are very similar to variables and anonymous functions, except that they are assigned to the object by using the dot notation (`.`). JavaScript is a dynamic language, and properties and methods are created by being assigned. The same notation is used to read the properties and execute the methods. The following example creates an object called "car", and adds a property and a method.

```
var car = new Object();
```

```

car.speed = 0; // the speed property is created and initialized
// the drive method is assigned an anonymous function
car.drive = function() {
    this.speed = 60; // from 0 to 60 in a few nano-seconds isn't bad
}
console.log(car.speed); // prints 0
car.drive(); // execute the drive method
console.log(car.speed); // prints 60

```

The this Keyword and the Global Object

Note the use of the **this** keyword in the drive method. If the keyword was omitted, JavaScript would look for a variable called "speed" on the global object, instead of the "car" object. The global object is the browser window, so the following two assignments are equivalent. They both create and/or assign the value to the speed property on the global object. Whenever an object refers to any of its members, it must use the **this** keyword to reference them.

```

car.drive = function() {
    speed = 60;
}
car.drive = function() {
    window.speed = 60;
}

```

In fact, the variables declared in the previous topics have all been declared on the global object.

```

var enigma = 42;
console.log(enigma); // prints 42
console.log(window.enigma); // prints 42

```

The same is true for functions.

```

var sayHello() {
    console.log("Hello");
}
sayHello(); // prints "Hello"
window.sayHello(); // prints "Hello"
sayHello === window.sayHello; // returns true

```

Literal Object Notation

JavaScript has a simpler, terser syntax for creating objects, called *literal object notation*. Instead of using new Object(), you can use braces. The following lines perform the same operation—each creates a new object.

```

var obj1 = new Object();
var obj2 = {};

```

Properties and methods can also be declared inside the curly braces. Remember to use colons (:) instead of equal signs (=) to assign the values. Multiple properties can be assigned by separating them with commas (,).

The following object is equivalent to the car object declared above.

```

var car = {
    speed: 0,
    drive: function() {
        this.speed = 60;
    }
}

```

```
};
```

Objects declared with literal object notation can be passed to functions, and functions can return them. The following example passes a literal object to the `summarize` function. The `summarize` function returns a new literal object.

```
function summarize(details) {
  return {
    name: details.firstName + " " + details.lastName,
    score: details.homeworkGrade * 0.6 + details.examGrade * 0.4
  }
}
var summary = summarize({
  firstName: "John",
  lastName: "Smith",
  homeworkGrade: 80,
  examGrade: 100
});
console.log(summary.name); // prints "John Smith"
console.log(summary.score); // prints 88
```

The JavaScript interpreter does not always parse code correctly. In the following example, the interpreter injects a semicolon after the `return` keyword because it mistakenly interprets the opening brace on the next line as a separate block. Calling this function returns `undefined`, instead of the expected object.

```
function getObject() {
  return // the interpreter injects a semi-colon right here
  {
    number: 1
  };
}
```

It is always safe to place the opening brace at the end of the previous line; the interpreter will not inject semicolons when they are on the same line. For consistency and to avoid similar pitfalls elsewhere, opening braces in JavaScript should always be placed at the end of the preceding line.

Functions Are Objects

Functions and objects in JavaScript have a unique relationship: at a very deep level, functions are objects. Whenever a function is created or declared, JavaScript actually creates a **Function** object, which derives from the **Object** object. This relationship helps explain why functions are values and not purely structural elements.

However, the relationship doesn't stop there. We've seen how simple custom objects can be created by either calling `new Object()` or using *literal object notation*. These objects can be useful, but they have a number of caveats: they are single-use objects and they cannot share behavior. Two literal objects that have properties and methods with identical signatures do not actually share their property and method definitions. The similarity is superficial.

In the following example, the `createCar` function creates an object with three members: one property ("speed") and two methods ("go" and "stop"). When a "car" object is created, the object's members are also created. Every member, including the method functions, is a value and the JavaScript engine has to allot it some memory. Because none of the member definitions are shared, every property and every method is created again for every "car" object. In addition, there is no real relationship between the two objects.

```
function createCar() {
```

```

var car = {}
car.speed = 0;
car.go = function() {
    this.speed = 60;
},
car.stop = function() {
    this.speed = 0;
}
return car;
}
var firstCar = createCar();
var secondCar = createCar();

```

Constructors

Note that the preceding example relies on the use of a factory method to create the objects. A factory method (**createCar**) is used to avoid repeating the code, which is good practice and should be encouraged. However, inside the factory method, the object is created by appending, or attaching, arbitrary methods and properties. The fact that the factory method is creating an object that represents a car is completely contextual. The problem is that the object is anonymous. Other than defining structure and behavior, classes also name the type of the object; so object creation relies less on the immediate context.

JavaScript supports creating objects that are not anonymous. The mechanism relies on the basic principle that functions are actually objects. Using the **new** keyword with *any function* instructs the JavaScript engine to create an object named after the function, and use that function as its constructor.

Creating a simple constructor in JavaScript is as easy as creating a function. Note that the first letter of the function's name is a capital letter. This is a common convention used to denote constructor functions, and it is considered best practice.

```

function Car() {
}

```

What makes a function a constructor is not the way it is declared, but the way it is called. To create an object, call the function with the **new** operator. This instructs the JavaScript engine to create a new object that uses the function as its constructor.

```

var firstCar = new Car();
var secondCar = new Car();

```

One way to add properties and methods to objects created in this way is to simply define them in the constructor, using the **this** keyword. We've seen earlier that the **this** keyword can be used to add variables and nested functions to a function. Since functions are actually objects, adding variables and nested functions is actually the same thing as adding properties and methods.

We can now replace the factory method used to create the anonymous "car" objects above with a **Car** object constructor:

```

function Car() {
    this.speed = 0;
    this.go = function() {
        this.speed = 60;
    },
    this.stop = function() {
        this.speed = 0;
    }
}

```

```
var firstCar = new Car();
var secondCar = new Car();
Constructors, just like every function, can also receive arguments:
function Car(currentSpeed) {
    this.speed = currentSpeed;
}
var firstCar = new Car(0);
var secondCar = new Car(30);
```

The instanceof keyword

Objects created using a function as a constructor are considered instances of the type named after the constructor. They can be said to share an “is” relationship with that type. To test for this relationship, use the **instanceof** keyword. Note that the keyword can be used to compare any value with any function; there must be a function (or an object) on the right side of the operator.

```
firstCar instanceof Car; // returns true
Car instanceof Object; // returns true because Car "is an" Object
Object instanceof Car; // returns false
```

What Is a Prototype?

In most object-oriented languages, classes are used to define the structure and behavior for objects, so every object of a given class has all of that classes' properties and methods. Classes also establish an inheritance hierarchy, so objects have all the properties and methods of their class *and* any class they inherit from. JavaScript uses prototype-based inheritance instead of class-based inheritance. In *prototype inheritance*, an object called a *prototype* serves as a template for other objects based on it. Another way to look at this is that instead of using a base *class* to define structure and behavior, JavaScript uses a base *object*, known as a prototype.

Every constructor function has a **prototype** property. By default, the value is undefined. When a prototype is assigned to a constructor function, all the objects created using that constructor function share the behavior and structure of the prototypical object.

```
function Car() {
    this.speed = 0;
}
Car.prototype = {
    go: function() {
        this.speed = 60;
    }
}
var theCar = new Car();
console.log(theCar.speed); // prints 0
theCar.go(); // calls the method defined on the prototype object
console.log(theCar.speed); // prints 60
```

If multiple objects are created by using the **Car** constructor function, they all share the same *instance* of the **go** method because it is defined on the prototype. The method is shared among all the derived objects. And because only one instance of the prototype and its members is created for all the derived objects, the JavaScript engine consumes less memory and performs better.

Adding properties to prototypes is tricky and should usually be avoided. First, since all the derived objects share the same instance of the prototype object, if any of them change the value of any member (property or method) of the prototype object itself, it affects all the other derived objects. So it only makes sense to add constant variables that are not expected to change the prototype.

Second, specifying properties on the prototype may produce unexpected results because of the way the **this** keyword works. When the prototype method **go** is executed in the following example, the **this** keyword refers to the nonexistent property on the actual object for which it is called, and not to the property defined on the prototype object. This causes the **speed** property to be created unexpectedly and can produce unpredictable results. It is strongly recommended to always specify variables in the constructor function, unless they represent constant values.

The built-in **hasOwnProperty** method can be used to test whether the property is on the actual object or on the prototype.

```
function Car() {
}
Car.prototype = {
  speed: 0, // not recommended
  go: function() {
    this.speed = 60;
  }
}
var theCar = new Car();
console.log(theCar.hasOwnProperty("speed")); // prints false
theCar.go();
console.log(theCar.hasOwnProperty("speed")); // prints true because the property has
been created on theCar, which is probably unintended
```

JavaScript is a dynamic language, so properties and methods can be added to a prototype object at any time. These changes affect all the derived objects. The following example adds a new method (**stop**) after a derived object has been created and then calls it.

```
function Car() {
  this.speed = 0;
}
Car.prototype = {
  go: function() {
    this.speed = 60;
  }
}
var theCar = new Car();
theCar.go();
console.log(theCar.speed); // prints 60
Car.prototype.stop = function() {
  this.speed = 0;
}
theCar.stop();
console.log(theCar.speed); // prints 0
```

Prototype Chains

Deeper hierarchies can be created by setting the prototype of a constructor function to another object with a prototype. Using a prototype chain, it is possible to create complex inheritance trees.

```
function Vehicle() {
  this.location = "";
  this.arriveAt = function(newLocation) {
    this.location = newLocation;
    console.log("Arrived at " + newLocation + ".");
  }
}
function Car() {
  this.driveTo = function(newLocation) {
    console.log("Driving down the road...");
  }
}
```

```

        this.arriveAt(newLocation); // call a method on the Car's prototype, its "base"
        object
    }
}
Car.prototype = new Vehicle(); // create the prototype chain
var car = new Car();
car.driveTo("New York"); // call a method directly on the Car object, prints "Driving
down the road...", "Arrived at New York."
console.log(car.location); // access a prototype's property, prints "New York"

```

Replacing the typeof operator

The **typeof** operator is used to check the type of a value (or its container). However, the **typeof** operator often produces counter-intuitive or inconsistent results that are difficult to use. For example, when the **typeof** operator is used with a primitive Boolean value, it returns the name of the primitive type: "boolean". But when it is used with a **Boolean** object, it returns the string "object". This difference is often at odds with expectations.

```

typeof true; // returns "boolean"
typeof new Boolean(true); // returns "object"

```

To achieve parity, another mechanism can be used. Instead of the **typeof** operator, use a function as shown in the following example.

```

function getType(value) {
    if(someValue === undefined) {
        return undefined;
    }
    else {
        return Object.prototype.toString.call(someValue).slice(8, -1);
    }
}
console.log(getType(true)); // prints "Boolean"
console.log(getType(new Boolean(true))); // prints "Boolean"

```

The **call** method used in **getType** causes the JavaScript engine to run the **toString** method for the **value** variable. The first argument to the **call** method is used as the value of the **this** keyword inside the **toString** method (or whatever method is called in this manner). This is a useful mechanism to override, or better control, the value of the elusive **this** keyword.

Built-In Objects

JavaScript has a number of built-in object types that provide very helpful services. The built-in **String**, **Number** and **Boolean** objects manipulate string, number and boolean values, respectively. In JavaScript, string, number and boolean types are not objects or classes. They are *primitive types*. The objects are often referred to as *wrappers* and provide helpful functionality for manipulating values of their respective types.

The constructors of the primitive wrapper objects are designed to be called with and without the **new** keyword. That is, they can be called as simple functions with a single argument, in which they convert the argument to their respective primitive value.

```

console.log(String(5)); // prints the primitive string value "5"
console.log(Number("5")); // prints the primitive numeric value 5

```

When they are called as constructors, with the **new** keyword, they create objects that can be used to manipulate their values. All three objects have a constructor that can accept an optional value. They evaluate to the empty value of their respective types if no value is passed to the constructor.

```
var str = new String("intriguing"); // wraps the string "intriguing"
var num = new Number(5); // wraps the number 5
var bool = new Boolean(); // wraps the Boolean false
```

All three objects have a **valueOf** method that returns their current value.

```
str.valueOf(); // returns "intriguing"
num.valueOf(); // returns 5
bool.valueOf(); // returns false
```

The object methods of the wrapper objects can be called on values or variables of the respective primitive types, so the objects are rarely used directly.

```
"intriguing".valueOf(); // returns "intriguing"
5.valueOf(); // returns 5
true.valueOf(); // returns true
```

The **String** object contains several functions that help manipulate string values, including case conversions (**toLowerCase** and **toUpperCase**), trimming functions (**trimLeft**, **trimRight** and **trim**, which clear whitespace from the left, right or both ends of the string), and many other string manipulations (for example, substring extraction and string replacement functions).

```
var str = new String("  too much whitespace  ");
str.trim(); // returns "too much whitespace"
```

The **Math** object provides many mathematical functions and constants, including the very popular **PI** and **e** constants and the **abs**, **ceil**, **floor**, **round**, **min**, **max** and trigonometric functions. Note that the **Math** object is unlike other objects: instances of the **Math** object cannot be created.

```
Math.floor(Math.PI); // returns 3
```

When the **Date** object is created, its default value is a number representing the current date and time. The object can be queried using its various get functions.

```
var date = new Date(); // contains the current date
date.getFullYear(); // returns the current year as a number
date.getHours(); // returns the current hour as a number between 0 and 24
```

It can also be initialized with a specific date by using one of the constructor variants, its value can be modified by using one of the set functions (**setFullYear**, **setHours**, etc.) and it can parse string values as dates.

The **Array** object in JavaScript has syntax similar to C#'s syntax for value assignment and retrieval using indexers.

```
myArray[5] = 3; // assigns the value 3 as the 6th item in the array
myArray[5]; // returns the value 3
```

However, other than the syntax similarity, it behaves very differently. JavaScript arrays can be created with a constructor.

```
var items = new Array();
```

They can be expanded by simply assigning new values at unused indices.


```
items[3] = "a new value"; // expands the array and assigns the new value to the array
at the specified index
```

They can be initialized with a default size, for performance, and the size can later be expanded without difficulty.

```
var items = new Array(3);
```

The array can also be initialized with values.

```
var items = new Array("a", "b", "c");
```

The items in the array can be of different types.

```
var items = new Array("a", 1);
```

An array can be declared with a simpler notation called literal notation.

```
var items = ["red", "yellow", "blue"];
```

In addition, the **Array** object exposes functions that can manipulate the array. For example, the **push** and **pop** functions expose stack-like functionality. The **concat** function joins two arrays. The following example demonstrates the **sort** function.

```
var items = [5, 4, 3, 2, 1];
items.sort(); // sorts and modifies the array
items; // returns the array [1, 2, 3, 4, 5]
```

To iterate over an array, use the **for** function, which has basic syntax identical to C#.

Scope

- Only functions have scope
- Function scope determines where properties and methods can be accessed from
- A function keeps its environment alive in a closure if:
 - It is returned from another function as a value
 - It references scoped members from the other function
- Members declared inside loops are *hoisted* up to the scoped function

```
function getSimpleFunction() {
  return function () {
    var text = "simple function";
    console.log(text);
  }
}
var simpleFunction = getSimpleFunction(); // the result of getSimpleFunction is another function
simpleFunction(); // prints "simple function"
```

Function Scope

Scope refers to the rules that govern where a variable or other construct can be referenced or assigned in the code and is used to determine how accessible any member property or method is to other callers.

In JavaScript, only functions determine scope. Because objects are actually functions, scope is shared by objects also. However, code blocks, such as those found in `for` and `while` loops, do not specify scope. This can be quite confusing. The following example illustrates this behavior.

```
function loop() {
  for(var index = 0; index < 10; index++) {
    var current = index;
  }
  console.log(index); // prints 10 because the loop increments after the last
iteration
  console.log(current); // prints 9, the last value visible inside the loop
}
loop();
console.log(index); // raises a ReferenceError because index is out-of-scope
```

Both the **index** and **current** variables are visible anywhere inside the function, including outside the code block. The reason for this is code blocks do not manage scope. The only construct that has scope in the example is the **loop** function.

Nested functions can see all the members defined in all the parent functions. The nested function can see all the variables defined in any of its outer scopes, including those defined on the global object, itself a function.

```
var globalValue = "global";
function parent() {
  var parentValue = "parent";
  function nested() {
    var nestedValue = "nested";
    console.log(nestedValue); // prints "nested"
    console.log(parentValue); // prints "parent"
  }
}
```

```

    console.log(globalValue); // prints "global"
    console.log(siblingValue); // raises a ReferenceError because siblingValue is
out-of-scope
  }
  function sibling() {
    var siblingValue = "sibling";
  }
  nested();
}
parent();

```

Closure

It is very common for functions to return other functions that can be called later. One way to do this is to simply return another function.

```

function getSimpleFunction() {
  return function() {
    var text = "simple function";
    console.log(text);
  }
}
var simpleFunction = getSimpleFunction(); // the result of getSimpleFunction is
another function
simpleFunction(); // prints "simple function"

```

Another way is to create an object with a method, which is really just a function, and return that object.

```

function getObjectWithMethod() {
  var text = "object with method";
  return {
    method: function() {
      console.log(text);
    }
  };
}
var objectWithMethod = getObjectWithMethod(); // the result of getObjectWithMethod is
an object that has another function as a method
objectWithMethod.method(); // prints "object with method"

```

In these examples, each of the outer functions defines a **text** variable and then creates a new function that uses the **text** variable. The new functions are executed *after* the outer functions exit. Both new functions successfully print out the values of their respective **text** variables when they are called, even though the **text** variables are no longer in scope.

This code works because of *closures*. When a function returns another function, and the second function references any variable defined on the first function, JavaScript creates a closure. A closure keeps the variables referenced by the new functions “alive” even after the original function that defined them is no longer around. Another way to put it is that a closure keeps the new function’s environment and that environment contains all the referenced variables, regardless of where they were defined.

It also does not matter how the new function is created. It can be returned directly by the outer function. It can be a method of a returned object. It can be a nested function inside a function that is returned directly. It can be a value in an array. To create a closure, any returned function simply has to reference a variable defined in an outer scope.

In the following example, the “outer” function returns an object that contains some functions that access the **value** variable defined on the outer function. The object itself does not have any properties, just methods that modify or return the **value** variable.

```
function getCalculatorFor(number) {
  var value = number;
  return {
    add: function(operand) {
      value = value + operand;
    },
    multiply: function(operand) {
      value = value * operand;
    },
    getValue: function() {
      return value;
    }
  };
}
var calculator = getCalculatorFor(10); // returns an object with several methods
console.log(calculator.getValue()); // prints 10
calculator.add(5);
console.log(calculator.getValue()); // prints 15
calculator.multiply(2);
console.log(calculator.getValue()); // prints 30
```

They can do this because all the functions share the same exact closure, and the same environment.

Hoisting

Only functions in JavaScript create scope. Code blocks, including loop statement, do not. This is why variables declared inside loops can be seen from anywhere in the function, including locations outside the loops.

```
function iterate() {
  for(var index = 0; index < 10; index++) {
    var current = index;
  }
  console.log(index); // prints 10, the result of the last increment after the last
iteration
  console.log(current); // prints 9, the result of the last iteration
}
iterate();
```

The **index** and **current** variables are visible outside the for loop because the for loop's code block does not create scope. Both variables belong to the scope created by the **iterate** function, which is why printing them did not produce a **ReferenceError**. The function can be rewritten as the following example shows.

```
function iterate() {
  var index;
  var current;
  for(index = 0; index < 10; index++) {
    current = index;
  }
  console.log(index); // prints 10, the result of the last increment after the last
iteration
  console.log(current); // prints 9, the result of the last iteration
}
iterate();
```

In fact, JavaScript goes a step further. Any variable declared *anywhere* in the function, not just inside loops and nested code blocks, is *hoisted*, or lifted up, by the interpreter and declared as early as it can be in the function. Note the difference between the **good** variable and the nonexistent **bad** variable in the following example.

```
function printValues() {
  console.log(good); // prints "undefined", because the variable is implicitly
  declared (because it is hoisted) and left unassigned
  var good = "Very Good!";
  console.log(good); // prints "Very Good!", because the previous statement assigned
  a value to the "good" variable
  console.log(bad); // raises a ReferenceError, because the "bad" variable is not
  declared anywhere in the function
}
printValues();
```

The first time **good** is printed, it has the value **undefined**, because it was hoisted: it is implicitly declared at the beginning of the function and as yet unassigned. The second time **good** is printed, it has already been assigned a value, so the value is printed. When **bad** is printed, a **ReferenceError** is raised because the **bad** variable has not been declared anywhere in the function. Hoisting only applies to variables that are actually declared in the function, regardless of where they were declared.

Hoisting also affects closures. When an outer function declares a variable and then creates and returns a function that uses the variable, the closure contains a reference to the variable. The important thing to note is that hoisted variables can also be referenced by the closure. Because the closure contains a reference to the variable, the new function always sees the last value assigned to the variable when the original function exited.

```
function outer() {
  var inner = function() {
    console.log(value);
  };
  var value = "Hello!";
  return inner;
}
var printHello = outer(); // contains the inner function
printHello(); // prints "Hello!"
```

The call succeeds even though the **inner** function is defined before the reference variable **value** is declared. There are two reasons for this: the **value** variable is hoisted so the **inner** function can "see" it; and although the actual variable value when **inner** is declared is **undefined**, the **inner** function is called long after **value** is assigned a value and the **outer** function exits.

The Document Object Model (DOM)

- The DOM is the object representation of the HTML document
- The **window** object is the global object and holds the **document** object
- The **document** object represents the **html** element
- DOM element objects expose methods to search for elements
- Elements can be added or removed from the DOM
- Elements can be modified through the DOM

```
var newHeader = document.createElement("h1");
newHeader.innerHTML = "This is the New Header";
document.body.insertBefore(newHeader, paragraphs[0]);
```

The DOM

JavaScript was designed to interact with the HTML on the page. However, HTML is text and might contain mistakes. HTML structure is also rather difficult to parse. On the one hand, it represents a hierarchical structure, similar to XML, but on the other hand, the HTML syntax is very loose and lenient and permits great variation in writing styles. For example, some elements do not need to be closed, so figuring out which elements are siblings and which are children can be tricky.

With these and other problems in mind, when browsers parse and render the HTML, they also create an object model called the *Document Object Model*, or simply, the *DOM*, that is easier for JavaScript to interact with. The DOM contains objects that represent all the HTML elements on the page. The DOM is also structured in a way that mirrors the hierarchical layout the browser constructed based on the HTML.

The DOM is also an interactive API, so modifying the DOM affects the rendered document. The DOM exposes all the elements, their attributes and their styles. It is also possible to remove elements from the DOM (which removes them from the rendered document and the display) and to add new elements to the DOM.

The window object

JavaScript is an embedded language hosted by the browser. The browser exposes a global object, the **window**, which is the most basic interface between the JavaScript code and the DOM. All the constructors for the built-in objects, events and APIs are actually defined on the **window** Object.

In fact, any functions or variables defined outside a function are actually implicitly members of the **window** object. Whenever a variable is declared without the **var** keyword, it is added dynamically as a **window** object property. The following example demonstrates these facts.

```
var obj = new Object();
obj === window.obj; // returns true because obj is a member of the window object
obj === this.obj; // returns true because the window object, the global object, has
the current scope
```

The document object

The **window** object exposes the **document** property, which represents the root of the DOM and provides access to all the elements on the page. The **document** property is the DOM analog of the root **html** element. The document hierarchy is composed of nodes that all share some behavior and common properties. The root node is no exception, so the **document** object has an array of attributes on the **attributes** property that represent the actual attributes of the `<html>` element. It has a **className** property that represents the **class** attribute, as well as **id**, **name** and **style** properties and a host of other properties that can be associated with every element or specifically with the `<html>` element. Most of these properties can be modified by simply assigning them new values.

Every node in the DOM tree contains a number of properties that help navigate the hierarchy. So every node also has the following properties, **firstChild** and **lastChild**, **childNodes** (an array of all the element's children), and **parentNode**.

The document object also has unique properties specific to it. The **head** and **body** properties, which represent the **head** and **body** elements, are two such examples.

Finding Elements

DOM nodes also contain methods that are used to navigate the DOM more quickly. Some of the more commonly used methods are methods that search for elements on the page: **getElementById**, **getElementsByName**, **getElementsByTagName**, and the new **getElementsByClassName**.

The **getElementById** method returns a single DOM element that has the id passed as an argument. Because the HTML standard requires the values of the **id** attributes to be unique in any given document, the method is designed to return just one element, or null if none are found. The remaining search methods all return arrays of elements matching the specified criteria.

The following JavaScript example uses this HTML document.

```
<html>
<head>
  <title>This is a sample page</title>
</head>
<body>
  <div id="header">This is the page header</div>
  <p>This is a paragraph</p>
  <p>This is another paragraph</p>
</body>
</html>
```

With the following JavaScript example.

```
var headerElement = document.getElementById("header"); // finds the header div and
assign it to the variable
headerElement.style.border = "solid 2px green"; // equivalent to specifying the
style="border: solid 2px green" on the element itself
var paragraphs = document.getElementsByTagName("p"); // finds all the p elements on
the page, add them to an array and store the array in the variable
for(var i = 0; i < paragraphs.length; i++) {
  paragraphs[i].style.backgroundColor = "orange"; // equivalent to setting just the
background-color CSS property on each p element
}
```

The example code shows how the find methods might be used to find and manipulate the DOM.



Note: The `getElementsByClassName` method is a new method introduced in HTML5. It has wide support among modern browsers, but older browsers will not support it.

Manipulating the DOM

As we saw in the above example, changing the styles of elements on the page is very easy when working with the DOM.

The DOM also has methods that let us add and remove elements from the tree. For example, we could remove the header **div** above using the **`removeChild`** method of the header's parent.

```
header.parentNode.removeChild(header);
```

We can also add new elements very easily. To create an element, we first have to use the **`document.createElement`** method and pass it the name of the element tag we want to create. The **`createElement`** method is a factory method that prepares the new element object for us. Once we have an element, we can add it by calling the **`appendChild`** or **`insertBefore`** methods. The former adds the new element as its last child and the latter adds it as a child wherever we specify. The following example creates an **`h1`** element and adds it before the first paragraph we retrieved earlier.

```
var newHeader = document.createElement("h1");  
newHeader.innerHTML = "This is the New Header";  
document.body.insertBefore(newHeader, paragraphs[0]);
```


Working with JavaScript

- Add JavaScript to the page using:
 - Inline JavaScript
 - The **script** element
 - JavaScript files
- Visual Studio supports code completion with IntelliSense and **reference** elements

```
<body>
<h1>The Header</h1>
<script type="text\javascript">
<!--
    var header = document.getElementsByTagName("h1")[0];
    header.onclick = function() {
        alert("You clicked on the header!");
    };
// -->
</script>
</body>
```

Adding JavaScript to the Page

Inline JavaScript

Aside from properties and methods, DOM also exposes *events*. Events are essentially special functions called when certain things happen that we would like to respond to. For example, we might want to show the user a message when the user clicks on an element. We can respond to these events by adding *event handlers*. An event handler is code that executes when the event occurs.

The first place an event handler can be specified is in the HTML itself. In the HTML for the element, we can add inline JavaScript code to handle the event.

```
<h1 onclick="alert('You clicked on the header!');">The Header</h1>
```

Technically, we can add as much inline code as we want, as long as semi-colons properly separate the statements. In practice, we want to avoid using inline JavaScript in this manner as much as possible. It makes the code difficult to read, maintain and debug. It also violates the *Separation of Concerns Principle*, the idea that document behavior (the JavaScript code) should be separated from the content (the HTML).

The script element

The **script** element is our next option. We can place **script** elements anywhere in the **head** or **body** elements. On the script's **type** attribute, we specify "text/javascript" to let the browser know what kind of code the **script** element contains. So our equivalent code might now look like the following example.

```
<body>
<h1>The Header</h1>
<script type="text\javascript">
<!--
    var header = document.getElementsByTagName("h1")[0];
    header.onclick = function() {
        alert("You clicked on the header!");
    };
// -->
```

```
</script>
</body>
```

We have to find the element and then add the event handler. In this case, the event handler is an anonymous function that executes the code.

We placed the **script** element at the end of the body because code in the **script** element is always executed as soon as it is interpreted, just like the HTML is rendered as soon as it can be, even if the document has finished loading. If we had placed the **script** element in the **head** element, it may not have found the **h1** element, because it may not have been parsed yet.

Finally, because the script is inside the **body** element, we have to surround the script code with an HTML comment. This prevents browsers that lack support for JavaScript or browsers that have JavaScript disabled from showing the content as though it were regular HTML. Browsers that know JavaScript know to ignore these comments and will never render the code as text on the page. The last line in the **script** element also uses a JavaScript comment (`//`) to prevent the browser from accidentally reading the closing HTML comment tag dashes as the subtraction operator.

Script Files

Placing the code in a script element is certainly preferable to using inline code. But the script element only belongs to one page and cannot be reused across multiple pages. It is also difficult to organize and maintain JavaScript code interspersed in HTML files. We still haven't achieved complete separation of content and behavior. The best option is to put all the JavaScript code in separate files.

Instead of writing the JavaScript code inside the **script** element, we write the code in a separate file. To tell the browser to load the JavaScript file, we add the **src** attribute, which specifies the JavaScript file URL. This instructs the browser to load the JavaScript code from that file. The browser will not finish rendering the page until it loads and executes that JavaScript code, so it still runs exactly the same as before.

So the **script** element in the HTML file now looks like the following example.

```
<body>
  <h1>The Header</h1>
  <script type="text/javascript" src="script.js"></script>
</body>
```

And the JavaScript file **script.js** now contains only the JavaScript code.

```
var header = document.getElementsByTagName("h1")[0];
header.onclick = function() {
  alert("You clicked on the header!");
};
```

The code is identical, except that we no longer need HTML markup or the HTML comments. Only browsers that support JavaScript download the file and execute it.



Best Practice: Separating all our JavaScript into separate files is a best practice.

IntelliSense

Visual Studio supports IntelliSense for JavaScript. However, since JavaScript is a dynamic language much of the metadata the IntelliSense feature uses to discover the available objects, properties, methods, and other syntactic elements is simply unavailable. Visual Studio does a reasonable job of estimating with

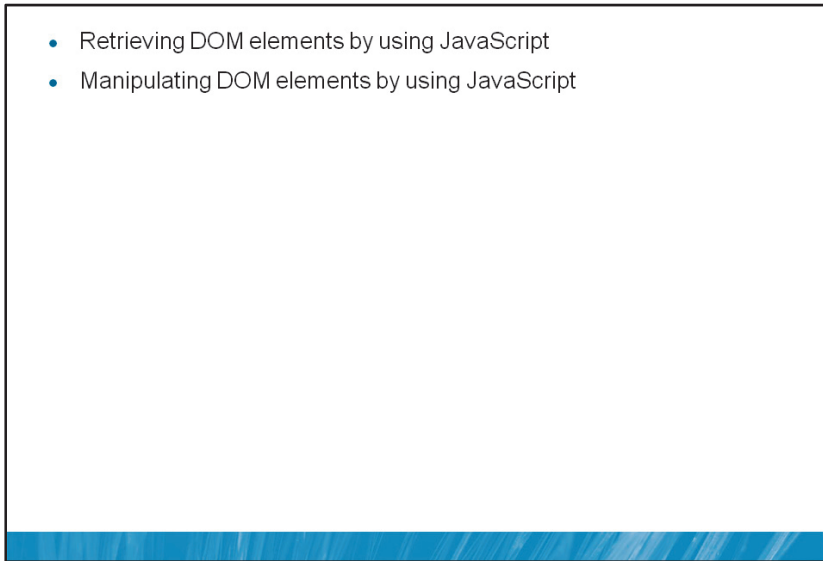
certain code constructs and objects. For example, it may have good knowledge of some of the built-in objects, like the **Math** object, but not for custom objects we create.

To help Visual Studio, we can add special comments to the beginning of our JavaScript code files that tell Visual Studio where it can find related JavaScript code. Some libraries, like the jQuery library which is introduced in the next lesson, have very detailed documentation that Visual Studio can also show us. The comments are only intended to help the IDE; they have no effect on the code itself or on the browser. To add a reference to another JavaScript file, add a comment with three forward slashes (instead of two) to the beginning of a JavaScript file. Add a **reference** element inside the comment with the path of the other JavaScript file, as the following example shows.

```
/// <reference path="jquery-1.5.1-vsdoc.js" />
```

As many lines as needed can be added. This instructs Visual Studio to use the especially marked-up jQuery documentation code file.

Demonstration: How to Work with the DOM by Using JavaScript



This demonstration shows how to use JavaScript to:

- Retrieve an element.
- Change the content of an element.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file, add HTML content to the **body** element.

```
<h1>The Header</h1>
<div id="content">The Content</div>
```

4. In the HTML file, add a **script** element inside the **head** element before the **head** element's closing tag.

```
<script type="text/javascript">
  function contentLoaded() {
    var content = document.getElementById("content");
    content.textContent = "This content was modified using JavaScript.";
  }
  document.addEventListener("DOMContentLoaded", contentLoaded, false);
</script>
```

5. Save the HTML file.
6. Run the saved file.
7. Observe that the webpage is showing the sentence, "This content was modified using JavaScript", instead of the original content.

Ajax and JSON

- Ajax improves the user experience by avoiding refreshing pages
- The **XMLHttpRequest** object communicates with the web server in the background using JavaScript
 - The **open** and **setRequestHeader** methods configure the request
 - The **send** method sends the request
 - The response is processed using the **readystatechange** event handler
- JavaScript Object Notation (JSON) is a terse format for sending JavaScript objects as structured data
 - Use **JSON.stringify** to convert a JavaScript object to a JSON string

```
var xhr = new XMLHttpRequest();
var product = {
  "id": 3,
  "isInStock": true,
  "colors": ["blue", "aqua", "green"]
};
```

Overview

Every time a user navigates from one page to another, the browser sends data to the web server and the web server responds by sending data back to the browser, including the HTML to render. Most of the data sent to the web server is metadata that rarely changes from page to page. In addition, because pages on a single site tend to share a common look and feel, much of the HTML sent back to the browser by the server is similar to that already sent for other pages on the same site. Repeatedly sending all of this redundant data wastes both bandwidth and time, which can be costly for both the user and the website's owners and detrimental to the user experience.

In addition, navigating from page to page, or refreshing a page (by pressing F5, for example), often clears the browser window before showing the new page. This can be disconcerting to users and does not provide an optimal user experience.

Ajax is a mechanism that lets the web developer use JavaScript to communicate with the web server directly. It can reduce the amount of data sent to the web server and back, and improves the user experience by allowing the developer to keep updating the page dynamically rather than requiring the user to navigate to a different page or refresh the page.



Note: The name Ajax was originally coined as an acronym, "AJAX," that stood for "Asynchronous JavaScript And XML". However, the technology has evolved over time, so now Ajax:

Can be used both synchronously and asynchronously.

- Does not necessarily require JavaScript (it can be used with VBScript, for example), though it is still used with JavaScript almost exclusively.
- Usually uses JSON (which will be covered shortly), instead of XML.
- As a result of these changes, it is now most commonly spelled "Ajax" instead of "AJAX."

Normal Page Retrieval Process

To understand how Ajax works, it is necessary to understand how a page is ordinarily retrieved. When the user clicks a link on a webpage or types a URL into the browser's address bar, the browser:

1. Determines which web server can process the URL.
2. Sends a message, known as a *request*, to the web server that instructs the web server to retrieve the webpage at the specified URL.
3. Waits for the web server to process the request and send a *response*, a message containing the HTML to display, back to the browser.
4. Processes the HTML content and displays the webpage.

The request is a message the browser sends to the web server, and the response is a message the web server sends back to the browser. Both messages have two parts: a set of HTTP headers, and a message body.

- Each HTTP header consists of a name and a value that describes the message body or instructs the web server or browser how to handle the message. Headers are used to specify the *MIME type* and encoding of the message, caching instructions, authentication information, and other metadata.
- The message body can be almost any data. In the normal page retrieval process outlined above, the message body is an HTML string. The message body is optional: most **GET** requests do not send a message body.

The most common HTTP header is the **Content-Type** header, which specifies the MIME type of the message body. A MIME type is a data format. The **Content-Type** header uses a two-part designation to specify the MIME type, written as a type name followed by a slash and a subtype. Some common MIME types include:

- text/plain – Plain text
- text/html – HTML
- text/css – CSS
- text/javascript – JavaScript
- text/xml – XML

MIME types can also designate binary data, audio and video formats, and other types of content. A full list of registered MIME types can be found on the IANA's MIME Media Types page at [MIME types](#).

During the normal webpage retrieval process, a lot of data is sent and received:

First, the request sent to the web server contains the requested URL as well as additional metadata, including but not limited to:

- The URL of the current webpage.
- All the cookies available to the web server.
- The supported languages and character sets.
- Authorization information.
- Information about the browser.

When the user submits a form, the same process takes place, except that the browser also sends the form data. Significantly, the web developer has almost no control over the data and metadata in requests sent

to the web server. The browser collects all the data and metadata and does not provide the developer an opportunity to intercept and modify the data sent to the web server.

Second, the response the server sends back to the browser contains metadata in addition to the HTML for the webpage.

When browsing within a single website, most of the metadata sent to the web server or returned to the browser does not change from request to request – but this data is repeatedly sent anyway. In addition, most websites use templates for all or most of their webpages to give them a consistent look and feel, which also means that much of the HTML content across the different pages is very similar or identical.

For example, in visiting the home page of Internet Explorer, a user would type the URL

http://windows.microsoft.com/en-US/internet-explorer/products/ie/home into Internet Explorer's address bar. Internet Explorer then:

1. Determines that the web server can be found at **windows.microsoft.com**.
2. Sends a request to **windows.microsoft.com** to retrieve the webpage at **/en-US/internet-explorer/products/ie/home**.
3. Waits for the server to send the response containing the HTML.
4. Processes the HTML content after the response arrives.
5. Displays the webpage.

This entire process is repeated for every link the user clicks on that webpage, including links to webpages on the same site.

Ajax was designed to circumvent the normal page retrieval process. It gives web developers greater control over the data sent to the web server, so a web server can be instructed to send only the minimum amount of data necessary to update an already open page.

The XMLHttpRequest object

Ajax is made possible by the **XMLHttpRequest** object, which is a component that lets JavaScript communicate with the web server in the background without refreshing the whole page. Because it operates in the background, users can continue to interact with the webpage while requests are sent and responses are received via the **XMLHttpRequest** object.

Most recent desktop and mobile browsers support the **XMLHttpRequest** object, so Ajax can be used on almost any device and platform.

The normal process for working with the **XMLHttpRequest** object is similar to the page retrieval process, except that the **XMLHttpRequest** object manages the process instead of the browser. One way to think about it is that the **XMLHttpRequest** object is like a "mini-browser" accessible via JavaScript. When JavaScript code needs to retrieve data from a URL in the background, it creates an **XMLHttpRequest** object that:

1. Determines which web server can process the URL.
2. Sends a request to the web server to retrieve the data from the specified URL.
3. Waits for the web server to process the request and send a response containing the requested data back to the **XMLHttpRequest** object.
4. Passes the data to a JavaScript function that can process the data and update the webpage.

While the **XMLHttpRequest** object waits for the web server to process the request (in step 3), the user can continue to interact with the webpage.

The use of an **XMLHttpRequest** object typically consists of three steps:

1. Creating and configuring the **XMLHttpRequest** object
2. Sending the request to the web server
3. Processing the response when it arrives

Creating and Configuring the XMLHttpRequest Object

Before an Ajax request can be sent, an instance of the **XMLHttpRequest** object has to be created. In browsers that support HTML5, creating the object is very simple.

```
var xhr = new XMLHttpRequest();
```

To configure the created object, specify the HTTP method and the URL address by calling the **XMLHttpRequest** object's **open** method.

```
xhr.open("GET", "/api/GetContent.aspx");
```

The first argument is the HTTP method, the actual command sent to the web server that designates the type of operation to perform. For simple uses, the **GET** method is typically used to retrieve data, and the **POST** method is used to submit form data.



Note: The HTTP method, also known as a *verb*, is the same used to specify the **method** attribute of the **form** element covered previously, except that additional verbs are supported by the **XMLHttpRequest** object besides just **GET** and **POST**. The other verbs, however, are beyond the scope of this course.

The second argument is the URL address where the requested content can be found. It can be the address of almost anything that can return data, such as another webpage, a web service, a text or XML file, or even a binary file.



Best Practice: Either a relative or absolute URL can be specified, but the browser may block requests to URL addresses located on other web servers as a security precaution. There are mechanisms to allow communication with other web servers, but they are beyond the scope of this course.

Addresses on the Web may change over time, and code for one site may also be used on another site, so web developers should avoid embedding the current known web server address in URLs found in JavaScript, HTML, or server-side code. Instead, web developers should use relative URLs, such as **scripts/myscript.js**, or rooted URLs that start with a slash (/), such as **/api/GetContent.aspx**. When the URL is a relative or rooted URL, the browser automatically sends the request to the current website's web server.

The **GET** method is very often used to search and filter information. When searching for information, a query string is typically appended to the URL. For example, a request to retrieve products in a specific category could look like the following statement. Note the use of the question mark (?) to separate the query string (category=Games) from the rest of the URL.

```
xhr.open("GET", "/Products/Search?Category=Games");
```


The **POST** method is most commonly used to submit forms. Form data can be sent in a query string, like the filter sent with the **GET** method, but query strings are inherently insecure because they are part of the publicly visible URL. In addition, forms may contain files and other binary data that may not be convenient or easy to send in a query string. Instead, form data is almost always sent as the message body.

Before a form can be submitted using the **XMLHttpRequest** object, the web developer has to add to the request metadata specifying that the data it contains represents form data. Metadata for HTTP requests is specified using headers that provide information about the message body. When an ordinary form is submitted, the browser automatically adds a few headers to the request. But when a form is submitted using Ajax, the web developer has to add the headers manually.

To add a header, use the **XMLHttpRequest** object's **setRequestHeader** method. The method accepts two arguments: the name of the header and its value. The following example adds the proper headers to designate the request as a form submission.

```
xhr.open("POST", "/Products/Create");
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
```

The **Content-Type** header describes the format of the message body. In this case, the format is the **x-www-form-urlencoded** MIME type, which is the MIME type associated with ordinary forms.

Sending the Request

After the request is ready, it can be sent by using the **XMLHttpRequest** object's **send** method. Requests sent using the **GET** method usually have no message body, so the method is called with no arguments.

```
var xhr = new XMLHttpRequest();
xhr.open("GET", "/Products/Search?Category=Games");
xhr.send();
```

Requests sent using the **POST** method usually send a message body, which is passed as an argument to the send method. When forms are submitted, the form data is serialized as (converted to) a query string (without the initial question mark), which is passed to the send method, as in the following example.

```
var xhr = new XMLHttpRequest();
xhr.open("POST", "/Products/Create");
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xhr.send("Name=My New Game&Category=Games");
```

This example sends form data serialized as a query string, but any data can be sent so long as the web server knows how to process it.



Best Practice: The **open** method actually has a third, Boolean argument called **async** that indicates whether the request is sent synchronously or asynchronously. The **async** argument is optional and defaults to **true**, so when the argument is omitted, as in the provided examples, the request is asynchronous.

When the request is asynchronous, the **send** method does not block the execution of code that comes after it, even if the response has not yet arrived.

- When the request is synchronous (when **async** is **false**), the **send** method does not end until the response has arrived.
- Avoid creating blocking synchronous Ajax requests, which can cause the UI to behave erratically and may be detrimental to the user experience.

Processing the Response

The **XMLHttpRequest** object has a property called **readyState** that determines the state of the request. The **readyState** property can have any one of the following values:

- **XMLHttpRequest.UNSENT** (0) – Designates that the object has been created but not yet been configured, and that the request has not yet been sent
- **XMLHttpRequest.OPENED** (1) – Designates that the open method has been called, but that the request has not yet been sent
- **XMLHttpRequest.HEADER_RECEIVED** (2) – Designates that the request has been sent and that the response headers (the metadata) have been received, though the response message body has not fully arrived yet
- **XMLHttpRequest.LOADING** (3) – Designates that the response message body, which can be large, is currently being received
- **XMLHttpRequest.DONE** (4) – Designates that the response message body has fully arrived



Best Practice: When the value of the **readyState** property is checked to determine the current state of the **XMLHttpRequest** object, it is possible to compare the state to the number or to the constant. However, the constant is recommended because it makes the code more readable.

For example, to check if the response has finished arriving, avoid using the following code.

```
if(xhr.readyState === 4)
```

Instead, use the following code.

```
if(xhr.readyState === XMLHttpRequest.DONE)
```

Whenever the **readyState** property value changes, the **readystatechange** event occurs. To process the response when it arrives, add an event handler for the **readystatechange** event, check if the **readyState** property value is **XMLHttpRequest.DONE**, and then load the response data and act upon it. The following example checks the result of the product creation request and shows a dialog box with the new product's ID. For this example to work, a web server has to be able to process the request and return a response, so simply executing the example as-is without a web server is insufficient.

```
var xhr = new XMLHttpRequest(); // readyState == XMLHttpRequest.UNSENT
xhr.open("POST", "/Products/Create"); // readyState == XMLHttpRequest.OPENED
xhr.setRequestHeader("Content-Type", "application/x-www-form-urlencoded");
xhr.onreadystatechange = function() {
    if(xhr.readyState === XMLHttpRequest.DONE && xhr.status === 200) {
        alert("The new product ID is: " + xhr.responseText);
    }
};
xhr.send("Name=My New Game&Category=Games");
```

In the example, an anonymous function is assigned to the **readystatechange** event, making the function the event handler. When the **readystatechange** event occurs, the function is called. However, the function is called for every change, not just when the response finishes arriving. The **if** statement determines whether the response has finished arriving by checking if the **readyState** property value is **XMLHttpRequest.DONE**. It also checks if the request and response were completed successfully by checking the HTTP status – the value 200 indicates a successful round trip. Finally, the **XMLHttpRequest** object's **responseText** property contains the result produced by the web server.

There are a number of properties that can be used to retrieve the response or its metadata, depending on the code's requirements. The following list enumerates some of them:

- **responseText** – Returns the content of the response, formatted as text
- **responseXML** – Returns the content of the response, formatted as XML and loaded into an XML document object
- **responseType** – Returns a string that designates the type of data stored in the response
- **response** – Returns the content of the response formatted according to the value of the **responseType** property
- **status** – Returns the HTTP status code for the request

Just as almost any data can be sent to the web server, almost any data can also be sent back. The **responseType** property is a short string that designates the type of data. The **responseType** property can have one of several values, but the most significant is **json**. When the **responseType** is **json**, the content of the response property is formatted using a special syntax that can be used to construct objects, allowing the server to effectively send the client a structured object, instead of text that has to be parsed.

JSON

JSON is an acronym for **JavaScript Object Notation**. It is a special format for representing objects and is based on the literal object notation covered earlier. JSON is a lightweight format for representing structured data, designed for sending such data to the web server and back easily.

Before JSON was used, XML was the most common format for sending structured data with the **XMLHttpRequest** object. However, XML is a verbose format, which means that it requires more space and bandwidth to represent the same data.

The following XML document represents a typical product.

```
<product id="3">
  <name>My Game</name>
  <category>Games</category>
  <isInStock>true</isInStock>
  <colors>
    <color>blue</color>
    <color>aqua</color>
    <color>green</color>
  </color>
</product>
```

Using literal object notation, a **product** object representing the same data can be declared in a fashion that is terser and easier to read.

```
var product = {
  id: 3,
  name: "My Game",
  category: "Games",
  isInStock: true,
  colors: [ "blue", "aqua", "green" ]
};
```

JSON is a subset of literal object notation specially designed for transport. Unlike literal object notation:

- JSON does not support functions.
- JSON property names must be enclosed in double quotes ("). Single quotes (') are not supported.

The example **product** object would be declared by using JSON as the following example shows.

```
var product = {
  "id": 3,
  "name": "My Game",
  "category": "Games",
  "isInStock": true,
  "colors": [ "blue", "aqua", "green" ]
};
```

Programmers usually do not directly create or modify JSON objects. They use normal JavaScript objects and convert them to JSON strings before sending them to the web server. JSON strings sent back from the web server are immediately converted back to JavaScript objects. To convert a JavaScript object to a JSON string, use the **JSON.stringify** method.

```
var json = JSON.stringify({
  id: 3,
  name: "My Game",
  category: "Games",
  isInStock: true,
  colors: [ "blue", "aqua", "green" ]
});
console.log(json); // will print '{"id":3,"name":"My
Game","category":"Games","isInStock":true,"colors":["blue","aqua","green"]}'
```

To convert a JSON string back to a JavaScript object, use the reverse function, **JSON.parse**.

```
var product = JSON.parse(json);
console.log(product.name); // will print "My Game"
```

When communicating with the web server, using the **XMLHttpRequest** object, JSON is the most common format used to transfer information. The following example uses the **XMLHttpRequest** object to send a JSON string to the web server and process the JSON response.

```
var newProduct = {
  name: "Awesome Game!",
  category: "Games"
};
var xhr = new XMLHttpRequest();
xhr.open("POST", "/Products/Create");
xhr.onreadystatechange = function() {
  if(xhr.readyState === XMLHttpRequest.DONE && xhr.status === 200) {
    if(xhr.responseText === "json") {
      var product = JSON.parse(xhr.responseText);
      alert("The new product ID is: " + product.id);
    }
  }
};
var json = JSON.stringify(newProduct);
xhr.send(json);
```

The code in the example never directly edits the JSON string. It converts the **newProduct** object to a JSON string immediately before sending it to the web server for processing, and converts the resulting JSON string to the **product** JavaScript object immediately after the response arrives.

Lesson 5

Introduction to jQuery

- What Is jQuery?
- jQuery Syntax and Major Concepts
- Getting started with jQuery
- jQuery Selectors
- jQuery and DOM
- jQuery and Events

The jQuery JavaScript library is undoubtedly the most popular JavaScript library today. It is widely used in websites and web applications because of the way it simplifies the use of JavaScript. jQuery provides useful alternatives for common client-side development tasks, creating functionality which may not be available or cumbersome to use within the JavaScript language. It is also a cross-browser JavaScript library which means that the developer writes code once and targets all the major browsers.

In this lesson you learn about jQuery, the major jQuery concepts, and how to use jQuery for common JavaScript functionality.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the basic jQuery principles used in the course.
- Explain jQuery major concepts.
- Use jQuery selectors..
- Apply jQuery functions on the DOM.
- Wire JavaScript events by using jQuery.

What Is jQuery?

- Introduced in 2006
- Open-Source JavaScript library
- Cross-Browser support
- Provide useful alternatives for
 - Common JavaScript development tasks
 - Creating cumbersome functionality to use within the JavaScript language

What is jQuery?

jQuery was first announced by John Resig in January, 2006. Since then, jQuery has grown to be the most popular JavaScript library in the JavaScript development world. In 2009 Microsoft announced its support for jQuery development and added the library to most web application templates in Visual Studio 2010.

jQuery includes a very big eco-system which is growing rapidly and consistently. The main functionality that jQuery provides is alternatives for common JavaScript development tasks and helps to create functionality that without it would have been very hard to implement. During the writing of this course the currently available version of jQuery is 1.7.2 and it can be downloaded from the main jQuery site – <http://www.jquery.com>.

It is very important to understand that jQuery is not a substitute for knowing JavaScript nor is it a solution to all JavaScript problems. In order to master jQuery, web developers need to know how JavaScript works and how to use it correctly.

Why Use JQuery?

Consider the following questions:

- How do you locate elements with a specific class?
- How do you apply styles to multiple elements?
- How do you handle events in a cross-browser manner?
- How many hours have you spent dealing with cross-browser issues?

These questions and many other questions about JavaScript can help to understand the necessity of jQuery and the rationale for using it.

For example, locating specific class elements, before HTML5 introduced the `getElementsByclassName` function, was very cumbersome. To do so the developer needed to search in each DOM object to see if it contains the class attribute and if that attribute's value equals the given class name. Another example is

applying style to multiple elements that could include many JavaScript statements for such simple and expected functionality.

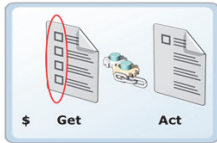
These questions also raise a very difficult issue in web development – the cross-browser issue. There are many available browsers and the old HTML4 specifications were not strict enough about how to implement JavaScript functionality. That led browser vendors to create their own functionality and made it hard to create web applications that interpret the same functionality in all browsers. For example, the `attachEvent` function in Internet Explorer 6 - Internet Explorer 8 vs. `addEventListener` function which was introduced in Internet Explorer 9 is a known cross-browser issue. Since most browsers used the W3C specifications with `addEventListener` and Internet Explorer versions below 9 did not, to address all browsers the developers needed to check what functionality was available in each browser before they could use it. This led to many problems and bugs and made web developers lives very miserable.

jQuery address the cross-browser issues using feature detection techniques. It checks whether a feature is available in the browser before using it. We discuss feature detection more thoroughly in Module 3.

jQuery Syntax and Major Concepts

- jQuery exposes a main object called jQuery
 - Also represented by the \$ sign
- Follows the Get >> Act philosophy
 - Select an HTML element and perform an action
`$(selector).action(...)`
 - Perform an action on the jQuery object
`$.action(...)`
- Every action returns the jQuery element for action chaining
`$(selector).action1(...).action2(...)`

```
$("#div").show().addClass("main").html("Hello jQuery");
```



jQuery syntax is very easy to follow. The jQuery object is represented by the word jQuery or \$ sign. The jQuery object is used to retrieve one or more elements using a specific selector and then performs chained actions on the selected elements.

For example, the following code retrieves all the div elements in the DOM and hides them.

```
$("#div").hide();
```

The same example can also be expressed as shown in the following example.

```
jQuery("div").hide();
```

jQuery uses the \$ and the jQuery notation to address conflicts with other frameworks which use the \$ sign. When a developer uses such a framework he can use the jQuery object instead of writing \$ and expose it through jQuery functionality.

Every jQuery action returns the jQuery object which enables the developer to chain actions and processes much more complicated functionality on the selected elements.

For example, the following code retrieves all the div elements and hides them all, except the first.

```
$("#div").slice(1).hide();
```

The developer can perform an action on the jQuery object itself, such as iterating an array and running some functionality on each element.

```
$.each(array, function (index, element) {
  ...
});
```

The three main concepts of jQuery are the Get >> Act philosophy, the \$() function, and its fluent API.

The Get >> Act philosophy

When using jQuery, the developer first selects an element and then performs actions on it, as shown on the previous slide. Using such philosophy helps to simplify jQuery use.



Note: Performing actions that result in reflow (rendering the webpage flow again because of some changes in its style or DOM) should be gathered together for better performance. This means if the code changes the DOM or its style then the developer should group these changes together statement after statement. Doing so helps the browser optimize the webpage changes and results in a smaller reflow process.

The \$() Function

The `$()` function is the core of jQuery. It exposes all jQuery functionality, including:

- Creating HTML elements on the fly.
`var newDivElement = $("

—creates a new div element in memory.`
- Manipulating existing DOM elements.
`$(window).width(300)`—changes the browser window width to 300 pixels.
- Selecting document elements using the jQuery rich selector API.
`$("div").hide()`—selects all the div elements in the DOM.
- Many other options such as AJAX functionality, DOM traversal, and DOM manipulation.

Fluent API

jQuery function names express their functionality, which makes for more readable code. Additionally, jQuery's functions return the jQuery object for each function call. This makes the code very expressive and the code "directs" its reader as to what it is doing. The following example can help to understand the concept.

```
$("div").show().addClass("main").html("Hello jQuery");
```

The code selects all the div elements in the webpage, displays them (if they are in hiding mode), adds a CSS class called `main` to each of them, and inserts into their HTML, the "Hello jQuery" string. The fluent API helps to understand the code and to determine what the developer wants to achieve with it.

Getting started with jQuery

- Download latest version from <http://jquery.com>
- Reference the script file in the web page
 - `<script type="text/javascript" src="jquery-1.7.2.js"></script>`
- Add jQuery IntelliSense
 - `/// <reference path="jquery-1.5.1-vsdoc.js" />`
- Wire the jQuery functionality after the document content is loaded

```
$(document).ready(function () {
    // perform the relevant action after the page is ready to use
});
```

To get started with jQuery first the developer needs to download its latest version from the jQuery site. Another option is to use Visual Studio 2010 web project templates, which include jQuery. The problem with this is that they add jQuery version 1.5.1 and the current jQuery version is 1.7.2. After retrieving the latest version of jQuery, it should be added to the webpage by using the script tag.



Note: Because jQuery is probably used in every page, it is good practice to put its declaration in Master Pages or in the main Layout. This is discussed in module 9.

After the developer has added the script, jQuery is available to use in the webpage. One other thing that can be very helpful is to add IntelliSense for jQuery. When downloading jQuery, the developer should also download the jquery-[VersionNumber]-vsdoc.js (it is added automatically in the Visual Studio Web project templates). This file includes documentation which Visual Studio is able to read and then add IntelliSense, which helps the developer. Adding IntelliSense is as simple as dragging the file into another JavaScript file or writing in the JavaScript file the following comment:

```
/// <reference path="jquery-1.5.1-vsdoc.js" />
```

After the developer has added IntelliSense, the developer wires up the jQuery functionality. The appropriate place to do this is after the content of the page is loaded. To wire up to this event, jQuery exposes the `$(document).ready()` function.

The following example shows the use the ready function.

```
$(document).ready(function() {
    // perform the relevant action after the page is ready to use
});
```

The previous example is equivalent to writing the following shorter code example.

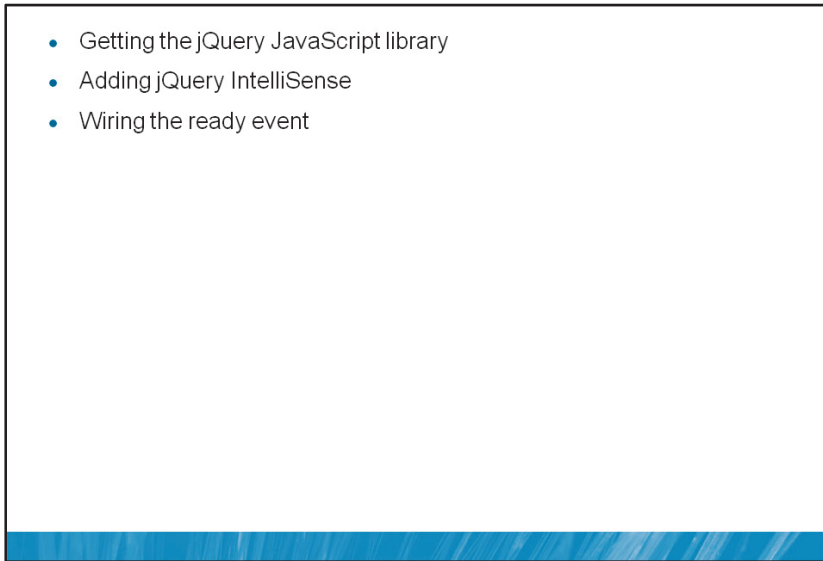
```
$(function() {
    // perform the relevant action after the page is ready to use
});
```

If the developer tries to implement some functionality before the content has loaded, it is not guaranteed to work because not all the DOM elements would be rendered. HTML5 exposes the `DOMContentLoaded` function, which is used widely in the course. Underneath, the jQuery `ready` function uses the same event, so the developer has the option to use each of the functions.



Note: Not all browsers support the `DOMContentLoaded` event (for example, all Internet Explorer versions under Internet Explorer 9 do not support this). jQuery uses feature detection and if the browser doesn't implement the `DOMContentLoaded` event, it uses other means to handle the ready event.

Demonstration: How to Get Started with jQuery



This demonstration shows how to:

- Add the jQuery script to the webpage.
- Add the reference to jQuery documentation for Visual Studio 2010 IntelliSense
- Wire the ready event and add some functionality.
- Run the webpage in the browser.

Demonstration Steps

1. Log on to the virtual machine named **SEA-DEV** as **Admin** with the password **Pa\$\$w0rd**
2. In Visual Studio 2010, create a new ASP.NET Empty Web Application called HellojQuery.
3. Add a new HTML file called HellojQuery.
4. Add a new JavaScript file called HellojQuery.
5. Add the jquery-1.5.1.js and jquery-1.5.1-vsdoc.js files from D:\Mod01\Democode\Assets.
6. In the HTML file, add the following script tags inside the head element.

```
<head>
  <title>Hello jQuery</title>
  <script src="jquery-1.5.1.js"></script>
  <script src="HellojQuery.js"></script>
</head>
```

7. Drag the vsdoc file into the surface of the opened HellojQuery.js file to create the statement.

```
/// <reference path="jquery-1.5.1-vsdoc.js" />
```

8. Add the following code in the HellojQuery.js file to demonstrate the wiring of the ready event.

```
$(document).ready(function() {
  alert("Hello jQuery ready function");
});
```

```
});
```

9. Set the HellojQuery.html as the startup page.
10. Run the application.

jQuery Selectors

- Selectors help to select specific page elements
- Single or multiple elements can be selected
- The selected elements are manipulated later with jQuery code
- Most jQuery selectors use CSS selectors underneath

```

$("div")
$("#someId")
$(".main")
$("div[id]")

```

One of the strengths of jQuery is its huge selectors API. This API is based mostly on CSS selectors.



Note: Some of the new CSS3 selectors were adapted from the jQuery selector API and now are starting to be implemented inside browsers as a specification.

The main purpose of jQuery selectors is to help detect and select specific page elements. The developer can use them to retrieve single or multiple elements wrapped inside the jQuery object. The selected elements are manipulated later by using jQuery API functions. The number of selectors combinations is enormous, enabling selecting very specific elements very quickly.

There are many options for selections. Some are described below:

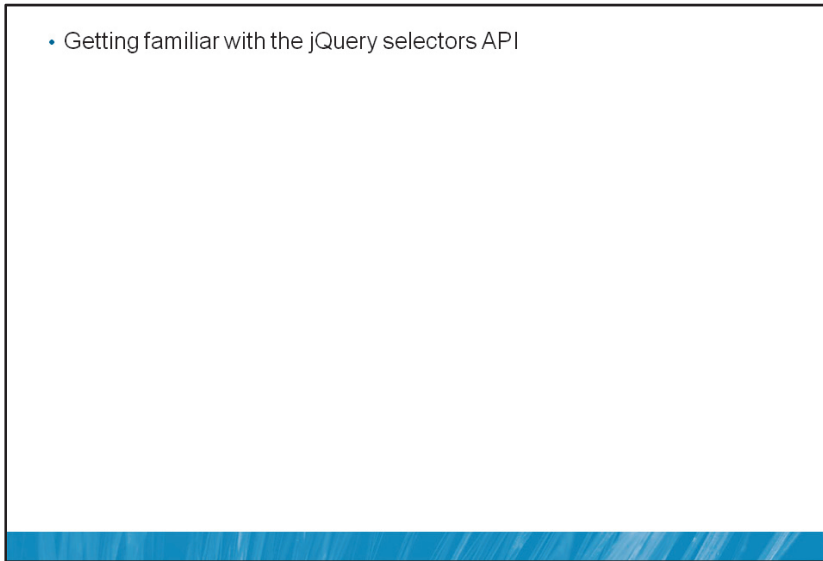
- Select all the elements in the webpage – \$("*").
- Select by a specific tag –
 \$("div") – select all the div elements in the webpage.
- Select by Id – use the # sign to indicate the id
 \$("#someId") – select the element with the someId id.
- Select by class – use the . sign to indicate a class name
 \$(".main") – select the element with a main class name.
- Precise or combination of selectors -
 \$("#content, .menu") – select the element with a content id that has a menu class name.
 \$("div.main") – select all the divs with a main class name.
- Hierarchy selectors -
 \$("table td") – select all the table td descendants
 \$("tr > td") – select all the td children of a tr
 \$("label + input") – select the input element next to a label element

`$("#content ~ div")` – select the div siblings of an element with a content id

- Attribute filters – use the `[]` signs to indicate an attribute
`$("div[id]")` – select all divs that have an id attribute
`$("div[dir='ltr']")` – select all the divs that have a dir attribute value equals to ltr
- Forms selectors
`$("input:checkbox")` – select all the inputs of type check box.
`$(":button")` – select all the buttons in the webpage

There are also special selectors such as the `:contains()` function, which can select elements that match the content within the contains parenthesis, the `:even`, `:odd` syntax that can select even or odd rows, `:first-child` that selects the first child of every element group and many more.

Demonstration: How to Use the jQuery Selectors API



This demonstration shows how to:

- Add markup to the webpage.
- Add the jQuery selectors functionality.
- Run the webpage in the browser.

Demonstration Steps

1. Log on to the virtual machine named **SEA-DEV** as **Admin** with the password **Pa\$\$w0rd**
2. Open the previous HellojQuery web application.
3. In the HellojQuery HTML file, in the body element, add a div with an id of main.

```
<div id="main">
</div>
```

4. In the main div, add the following code.

```
<ul id="ulOfItems">
  <li>item 1</li>
  <li class="red">item 2</li>
  <li id="item3">item 3</li>
</ul>
<br />
<div>Hello jQuery</div>
<div id="result"></div>
```

This creates the page markup containing an unordered list and two divs.

5. In the HellojQuery.js file, add the following code.

```
var resultDiv;
$(document).ready(function () {
  resultDiv = $("#result");
  $("li").each(function () {
```



```
        resultDiv.text(resultDiv.text() + " " + this.tagName);
    });
    $("#item3").each(function () {
        resultDiv.text(resultDiv.text() + " " + this.id);
    });
    $(".red").each(function () {
        resultDiv.text(resultDiv.text() + " " + this.tagName);
    });
    $("ul#ulOfItems").each(function () {
        resultDiv.text(resultDiv.text() + " " + this.id);
    });
});
```

Write to the result div, all tag names of the selected list items, the id of the selected item3, the tag name li once for the list item with the red class name, and the id of the unordered list, ulOfItems, retrieved by the precise selector.

6. Run the webpage in the browser to view the results.

jQuery and DOM

- jQuery has a variety of DOM traversal functions
 - Including iterating and finding DOM elements
- jQuery can create new HTML elements on-the-fly
 - The elements can be created in memory and appended to the DOM
- jQuery includes ways to manipulate the DOM
 - Make changes to element's attributes, set element's style and even modify the entire element

```
$("p").find(".header").show();
$("<p>My new paragraph</p>").appendTo("body");
```

After selecting HTML elements by using the jQuery selector API, the developer can traverse and manipulate these elements by using the jQuery API. These abilities can create an extremely powerful selection toolset.

DOM Traversal

jQuery includes a variety of DOM traversal functions. These functions help traverse the DOM tree and later implement needed functionality on them. It also allows acting upon multiple sets of elements in a single chain. For example, take a scenario of a webpage that includes some div elements.

```
<html>
  <head>...</head>
  <body>
    <div>...</div>
    <div>...</div>
  </body>
</html>
```

The developer can use the selectors API to retrieve all the div elements and then iterate them by using each function to perform an action.

```
$("div").each(function(i) {
    alert(i + "=" + $(this).text());
});
```

In the example, the result of running the code is two alerts with the index and the div's text.

There are many traversal functions in the jQuery core such as next, prev, siblings, children, parent and more. There are also functions that help to make deeper selections such as find, find elements inside a selected element, or slice, that returns a slice of the selected elements array according to the start and end index it receives.

The following example shows the use of the find function.

```
$("p").find(".header").show();
```

The code selects all the paragraphs and then finds elements with the class name "header" inside them.



Note: Many of the traversal functions can be replaced with a more specific selector. For example, the last code example, `$('#p').find('.header').show();` can be replaced with the following code.

```
$('#p.header').show();
```

DOM Creation

jQuery includes the ability to create the DOM element during runtime. To create an element, the developer passes a string that includes an HTML snippet to `$()`. jQuery attempts to create new DOM elements as described by the HTML snippet and returns a jQuery object that includes the created elements. The elements are created in memory and aren't appended to the DOM. It is the responsibility of the developer to append the new elements.

The following example creates a new paragraph and appends it to the body element.

```
$("<p>My new paragraph</p>").appendTo("body");
```



Best Practice: When you create a new element, on-the-fly does not append it to the DOM until you have finished creating it all and its functionality. Appending it to the DOM starts a reflow process, and if it is then manipulated again, another reflow process is created.

DOM Manipulation

jQuery includes many ways to manipulate the DOM. The developer can change an element's attributes, style, and even remove the entire element from the DOM. There are many functions in the jQuery API such as `html`, `text`, `val`, `empty`, and `remove` that help the developer achieve almost every JavaScript DOM manipulation. The following examples show the use of the manipulation API.

`$("#div").html("<div>Hello jQuery!</div>");` inserts a new div with the "Hello jQuery!" text inside every div in the webpage.

`$("#:text[name='txt']").val("Hello");` sets the value of a textbox with the txt name to Hello.

`$("#p").removeClass("blue").addClass("red");` replaces the blue class name to red in all webpage paragraphs.

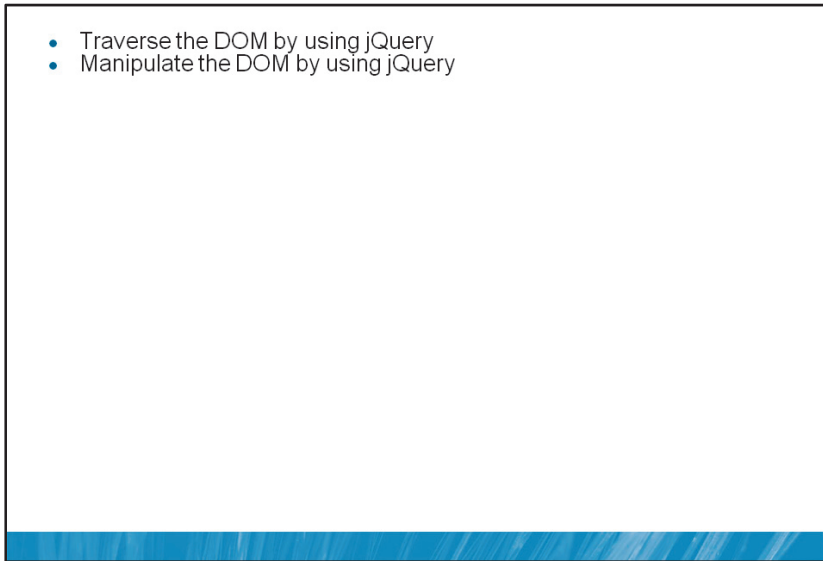
`$("#a").attr("href", "home.htm");` changes the attribute href to point to home.html in all webpage a elements.

Some of the functions such as `val` or `html` can be used also to fetch the content of the element. For example the following code fragment puts the value of the textbox with the txt name in the content variable:

```
var content = $("#:text[name='txt']").val();
```

It is definitely easier to use jQuery to traverse and manipulate the DOM. In contrast, plain JavaScript can be very cumbersome sometimes and can lead to writing a lot of code.

Demonstration: How to Work with the DOM by Using jQuery



This module shows how to:

- Use the previous demonstration.
- Add the traversal functionality.
- Add the DOM manipulation functionality.
- Run the webpage.

Demonstration Steps

1. Log on to the virtual machine named **SEA-DEV** as **Admin** with the password **Pa\$\$w0rd**
2. Open the previous HellojQuery web application.
3. Add the following button at the end of the HellojQuery.html file.

```
<input type="button" id="btnEach" value="Each Function Example" />
<br />
```

4. Add the following JavaScript code in the HellojQuery.js file inside the ready handler:

```
$("#btnEach").click(function () {
    $("div").each(function (i) {
        alert(i + "=" + $(this).text());
    });
});
```

The code wires a click event to the input with the button type. When the button is clicked, an alert is shown for each div in the page with the div index and the div text.

5. Run the webpage in the browser.
6. Click the button to run each functionality.
7. Add the following paragraph and buttons to the HellojQuery.html file.

```
<p></p>
<input type="button" id="btnCreateParagraph" value="Create Paragraph Data" /><br />
<input type="button" id="btnAppendToU1" value="Append Two List Items Using appendTo" />
```

8. Add the following JavaScript code in the HellojQuery.js file inside the ready handler..

```
$("#btnCreateParagraph").click(function () {
    $("p").html("<div>Hello jQuery!</div>");
});
$("#btnAppendToU1").click(function () {
    $("<li />").html("last").appendTo("u1");
    $("<li />").html("new first").prependTo("u1");
});
```

The code wires two click event handlers to the buttons. The first handler adds a new div element with the "Hello jQuery!" text to the paragraph element. The second event handler creates two list items that are added as the first and last list items in the unordered list in the HTML.

9. Run the webpage in the browser.
10. Click the buttons to run their functionality.

jQuery and Events

- Events notify their handlers that the user performed an action
- jQuery provides a cross-browser event model
- The jQuery event model is simple to use and provides compact syntax

```
$("#div").bind("click", function() {  
...  
});  
  
$("#div").one("click", function() {  
...  
});
```

Events are used to notify their handlers that the user performed an action. Every major language implements an event model to interact with users. Working with events in the past was not an easy task. Browsers implemented their own event models and functions—behavior that caused lots of headache for web developers. jQuery provides a cross-browser event model that can help reduce the complexity of using and wiring events. The event model is very simple and wiring events is as easy as telling an element that if it is clicked then it should run some handler. The following code fragment shows how to wire the click event to the element with a btn id.

```
$("#btn").click(function() {  
...  
});
```

The jQuery API includes a large number of events such as click(), blur(), focus(), dblclick(), mousedown(), mouseup(), and many more. Besides the event functions the developer can also use the bind function which executes always and the one function which executes the handler only once. Both these functions get as parameters the event name to wire to and the handler which the event executes. The following example shows the use of these functions.

```
$("#div").bind("click", function() {  
...  
});  
$("#div").one("click", function() {  
...  
});
```

Another option exposed to developers is to remove handlers by using the `unbind` function. The following example shows how to use it.

```
$("#div").unbind("click", function() {  
  ...  
});
```

The jQuery event model can simplify the JavaScript code to handle events. It can reduce errors caused by cross-browser issues regarding events.

Lab: Building a Basic Webpage in the Visual Studio 2010 SP1 Environment

- Writing the Home Page HTML
- Styling the Homepage
- Adding JavaScript and jQuery to Make a Dynamic Home Page

Logon Information

Virtual Machine	Virtual machine: SEA-DEV
User Name	User name: Admin
Password	Password: Pa\$\$w0rd

Estimated Time: 75 minutes

Scenario

You are informed that the management of Contoso, Ltd. has decided to create a new web application to replace the current one. This web application should follow the emerging standards of HTML5 and is expected to be a major milestone in the company's growth. The project manager supplies a design document written by the company's system analyst and a UI mockup created by the company's designer.

Objectives

After completing this lab, you will be able to:

- Use the Visual Studio 2010 SP1 environment.
- Use HTML, CSS, and JavaScript to create a basic webpage.

Exercise 1: Writing the Home Page HTML

Scenario

In this exercise, you will use the UI mockup image of the homepage created by the company's designer as a reference when you create the homepage structure.

► Task 1: Create a new Empty Web Application.

1. Open the starter solution, which is located in the **D:\Mod01\Labfiles\Starter** folder.
2. In the **GameStore** solution, create a new Empty Web Application called, **GameStore.Html**.

► Task 2: Create the solution structure.

1. In the **GameStore.Html** web application, add the directory structure and all images, which are located in the **D:\Mod01\Labfiles\Starter** folder.

► Task 3: Create the Homepage HTML file.

1. In the **GameStore.Html** web application, add a new HTML file called, homepage.

► Task 4: Create a simple HTML element.

1. In the **homepage.html** file, create a genre div and a game element, which includes the game image, the game title, the game price, and the button to add the game to the cart. The genre element should be created inside the div with **catalog** id. Use the provided designer image to understand how to do so.

► Task 5: Create a simple HTML table element.

1. Inside the div with **cart** id, locate the second and empty div and create a new HTML table element inside that div. Every row in the table will hold the game image, a game title, and an X image. Use the provided designer image to understand how to do so.

► Task 6: Create a simple unordered list.

1. Inside the div with **nav** class, create an unordered list that holds two list items. Each of the items should hold a link, the first to Home and the second to Checkout. Use the provided designer image to understand how to do so.

Results: A home page written in HTML containing a game catalog, a game in spotlight, and cart data.

Exercise 2: Styling the Homepage

Scenario

In this exercise, you will use CSS to apply styles and layout to the homepage, in accordance with the UI mockup.

► Task 1: Create the basic theme for the homepage.

1. Create a new CSS file and add the link element to the homepage.
2. Reset the default browser styles.
3. Add styles for common classes (.header, .button, .text, and so on.)
4. Lay out the logo (override the h1 style), the navigation bar, and the footer.

► Task 2: Lay out the main homepage elements.

1. Add the skeletal feature-specific CSS files to the project (slideshow.css, dragdrop.css, spotlight.css, catalog.css, and cart.css), which exist in the **D:\Mod01\Labfiles\Starter\Css** folder.

Results: A homepage with a layout and style in accordance with the UI mockup image document.

Exercise 3: Adding JavaScript and jQuery to Make a Dynamic Home Page

Scenario

In this exercise, you will use the homepage functionality document created by the company's system analyst to create the homepage behavior.

► Task 1: Create a new JavaScript file.

1. In the GameStore.Html web application, add the Scripts directory and a new JavaScript file called, GlobalFunctions.
Add a script reference to that file in homepage.html.

► Task 2: Add jQuery files and jQuery IntelliSense.

1. Add the jQuery files to the **Scripts** folder. Add the reference directive inside the **GlobalFunctions** file to include IntelliSense. Add a script reference to jQuery in **homepage.html**.

► Task 3: Create simple JavaScript objects.

1. Create the **Cart** and **Game** JavaScript objects.

► Task 4: Add event listeners to the JavaScript file.

1. Add an event listener to the DOMContentLoaded event.

► Task 5: Create the collapsible panel functionality.

1. Create the functionality of the collapsible panels for the genre div elements. When the webpage loads, collapse all the genres panels. Use the functionality document to understand the functionality demands.

► Task 6: Create the slideshow functionality.

1. Create the slideshow functionality to replace the images displayed in the div with a slideshow id. Use the functionality document to understand the functionality demands.

Results: A home page containing dynamic parts such as collapsible panels and an image slideshow, manipulated by using JavaScript and jQuery, as described in the design document

Module Review and Takeaways

The module discussed the three building blocks of every web application that you will build: HTML, CSS, and JavaScript. HTML is used to structure the webpage by using simple tag language. CSS is used to style and layout the webpage by using selectors and a variety of properties that you can apply on HTML elements. JavaScript is used to manipulate the Document Object Model and to create better user experience by making the webpage dynamic.

In addition, you were also introduced to jQuery, one of the most popular JavaScript libraries. jQuery helps to simplify repetitive JavaScript work. Knowing how to use jQuery can help you write less JavaScript code and be more productive.

Tools

Tool	Use for	Where to find it
jQuery	Finding objects and functions for querying and working with HTML pages using JScript.	Included with Web application projects in Microsoft Visual Studio 2010. Also at http://www.jquery.com

Module 2

Creating Page Structure and Navigation

Contents:

Module Overview	2-2
Lesson 1: HTML5 Structural Elements	2-3
Lesson 2: Navigation and Menus	2-16
Lab: Creating Page Structure and Navigation	2-30
Module Review and Takeaways	2-33

Module Overview

- HTML5 Structural Elements
- Navigation and Menus

The World Wide Web is constantly evolving. Every version of HTML introduced new elements and attributes to keep up with the demanding environment of the Web. HTML5 is not so different from HTML4 and introduces many new elements that help build better webpages. Those elements help to structure webpages and create sections in the webpage with more meaning than ever.

This module is about the new HTML5 elements. In this module, you will be exposed to the new elements and their meaning, and learn how and when to use them. The module also discusses navigation and menus, which are the backbone of creating a website or application. Without navigation, there won't be a way to move from one page to another; without menus, the website or application clients won't know the options that the site or application exposes to them.

Objectives

After completing this module, you will be able to:

- Review the new HTML5 structural elements.
- Examine the creation of web application navigation and the use of menus.
- Migrate HTML4 webpages to HTML5 webpages.

Lesson 1

HTML5 Structural Elements

- The Rationale Behind the New HTML5 Elements
- The New Structural Elements
- Semantic Block-Level Elements
- Inline Semantic Elements
- Interactive Elements
- Other New Elements

This lesson is about the new HTML5 structural elements. In this lesson, you will learn about the rationale of having so many new elements. After understanding the rationale, the lesson drills down into the new elements, which are organized into meaningful groups such as structural elements, block-level elements, inline elements, and interactive elements.

Lesson Objectives

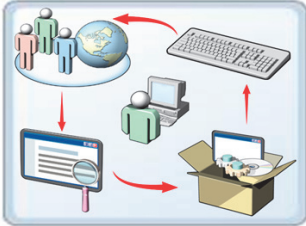
After completing this lesson, you will be able to:

- Explain the rationale behind the new HTML5 elements
- Describe where and how to use the new HTML5 structural elements
- Describe what are the new semantic block-level elements
- Describe what are the new inline semantic elements
- Describe what are the new interactive elements
- Describe other new elements

The Rationale Behind the New HTML5 Elements

New HTML5 elements enable:

- **Accessibility**
- **Searchability**
- **Internationalization**
- **Interoperability**



To understand why W3C created so many new HTML elements, we need to understand the World Wide Web. The World Wide Web is a collection of billions of websites and applications. Every website or application is made of textual and/or media content. In the past, to lay out a webpage, developers created container elements (mostly `div` elements) and repeated some patterns in creating these containers. For example, searching inside the source code of some websites might reveal that most of them have a **`div`** element with a **header** id or class name. HTML5 introduces the **header** element to replace the way developers wrote webpages in the past. Like the header, a lot of other semantic elements were added to the HTML5 specification. These elements aren't merely nice to have—they should be used whenever possible. The problem with that is that when the browser encounters an element that it does not recognize, the element simply won't be rendered. The problem can be solved by using CSS or external JavaScript libraries to fill this gap. You will learn how to do that later in the course.

The new elements enable accessibility, searchability, interoperability, and internationalization, thus making their use important. For example, an **article** element indicates to search engines that the content inside the element is an article, helping the search engine understand what the developer wants to express in the webpage. Webpages that use the new elements are also more open to people with accessibility problems, directing them quickly and easily to the main content.

The New Structural Elements

- New structural HTML5 elements:
 - Article
 - Footer
 - Header
 - Hgroup
 - Section
 - Address
 - Nav
 - Aside
- No need for **div** elements all over the webpage

```

<body>
  <header>
    <nav>
      <ul>
        <li>Navigation</li>
      </ul>
    </nav>
  </header>
  <section>
    <article>
      <p>Article content</p>
    </article>
  </section>
  <footer>
    <p>Copyright</p>
  </footer>
</body>

```



In previous versions of HTML, developers mostly used **div** elements to create structural elements in their webpages. Even though the use of **div** remains a valid way to structure a webpage, HTML5 introduces new structural elements that can play a more effective role. These new structural elements should be implemented in new webpages and applications, and even incorporated into already-existing ones. The elements help organize the content in a more coherent way, make the code more readable, and make the webpage more accessible.

The main structural elements are the following:

Article. Represented by the `<article>` tag. This element defines a self-contained composition. Articles are mostly used to present article content, such as blog entries, magazine articles, user comments, interactive widgets and gadgets, and any other type of independent content.

Footer. Represented by the `<footer>` tag. This element contains information about its containing element: its author, copyright information, and so on. Footers should be used at the lowermost part of a section or webpage. A webpage can have more than one footer element.

Header. Represented by the `<header>` tag. This element represents a group of introductory or navigation aids. Headers hold headings; navigation links and buttons; tables of contents, logos; and more. As with the footer, a webpage can contain more than one header element.

Hgroup. Represented by the `<hgroup>` tag. This element represents the heading of a section. It is used to organize heading elements, h1-h6, when there is more than one heading element one after another. This can happen in situations such as where there are subheadings or alternative titles.

 **Note:** **Hgroup** can hold only the heading elements. No other element should exist inside this element.

Section. Represented by the `<section>` tag. This element represents a generic section inside the webpage. The section will typically have a heading, but one is not strictly necessary. Sections help split webpages into logical sections, such as introductions, news items, carts, and so on.

Address. Represented by the `<address>` tag. This element represents contact information for its nearest article or body element ancestor. If the ancestor is the body element, the contact information applies to the whole document. Typically, the address element will be included in the footer along with other information.



Note: The **address** element must include only address information.

Nav. Represented by the `<nav>` tag. This element presents a collection of links to other pages for navigational purposes. The nav element will be discussed more thoroughly in lesson 2.

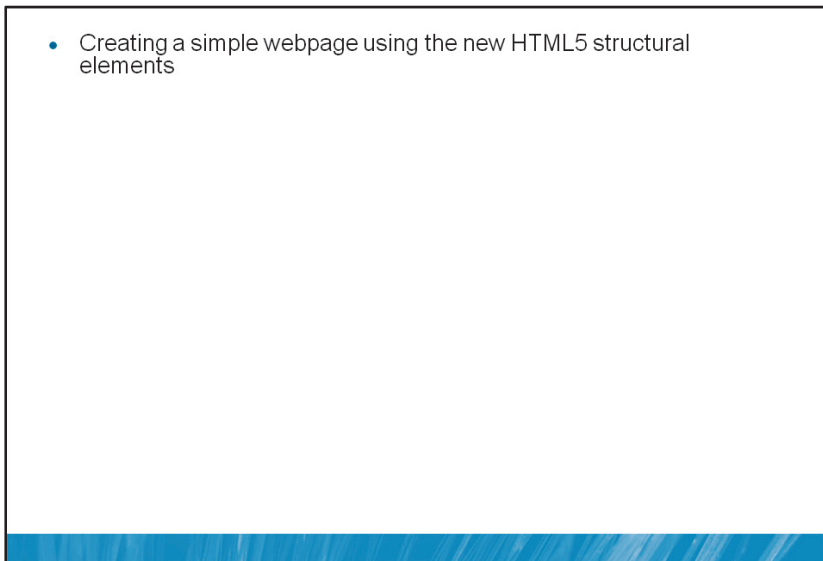
Aside. Represented by the `<aside>` tag. This element presents content related to the main content, such as notes, sidebars, or parenthetical remarks. Anything that won't change the meaning of the content if it is removed, but is related to the main content of the webpage, is a good candidate for the aside tag.

The following example of a simple webpage shows how to use these structural elements.

Webpage body element containing some structural elements

```
<body>
<header>
<h1>Page Title</h1>
<nav>
<ul>
<li>navigation 1</li>
<li>navigation 1</li>
</ul>
</nav>
</header>
<section>
<article>
<h2>Article Title</h2>
<p>Article content</p>
</article>
</section>
<footer>
<p>Copyright 2011 Microsoft</p>
</footer>
</body>
```

Demonstration: The New Structural Elements



This demonstration shows how to:

- Create a new Empty Web Application in VS2010
- Add a new HTML file
- Add markup to the web page using the new HTML5 structural elements

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file's body element add the following elements:

```
<body>
  <header>
    <h1>Page Title</h1>
    <nav>
      <ul>
        <li>navigation 1</li>
        <li>navigation 1</li>
      </ul>
    </nav>
  </header>
  <section>
    <article>
      <h2>Article Title</h2>
      <p>Article content</p>
    </article>
  </section>
  <footer>
    <p>Copyright 2011 Microsoft</p>
  </footer>
</body>
```

4. Save the file as an HTML file.
5. Run the saved file.

Semantic Block-Level Elements

Semantic Block-Level Elements

- Block elements that have a specific semantic meaning
- Include the figure element

```
<figure>
  
  <figcaption>Sports</figcaption>
</figure>
```



Along with the structural elements, HTML5 also introduces new block-level elements. Block-level elements are used inside structural elements to create a more expressive and semantic page layout. They typically contain inline elements or other block-level elements. HTML4 includes a lot of block-level elements such as `div`, `p`, `table`, and all the list elements.

Among the new block-level elements are:

Figures. Represented by the `<figure>` tag. This element specifies self-contained flow content, such as images or code. A common developer mistake is the impression that the figure element should be used to wrap images; rather, figures should be used to wrap illustrations, diagrams, photos, or code listings when these are referenced in the webpage content (such as when an article tells the reader to "see figure 1"). Figure elements can use the `<figcaption>` element to indicate a caption for the figure.

The following example contains a figure element.

Figure Element

```
<figure>
  
  <figcaption>Figure caption</figcaption>
</figure>
```



Note: You should not use **figure** elements to wrap images. Only if the images are referenced from some content should they be wrapped with a **figure** element.

Inline Semantic Elements

- Inline elements affect their content.
- Examples:
 - **Mark**
 - **Meter**
 - **Progress**
 - **Output**
 - **Time**

```

This is <mark>marked</mark> text
<meter value="70" min="0" max="100" low="54" high="100" optimum="100">70</meter>
Your download is <progress max="100" value="60">60%</progress> complete
Output: <input type="text" value="1" /> + <input type="text" value="3" /> =
<output>4</output>
The time is now <time datetime="2011-09-28T18:36">6:36 P.M. on November 28th</time>

```

This is **marked** text

The time is now 6:36 P.M. on November 28th

Inline semantic elements are used to indicate a specific and meaningful subsection of the content. Among this group of HTML4 elements, you can find the **** tag, which indicates bold text; the **<i>** tag, which indicates italic text; and many other elements.

The following examples show how to use the new inline semantic elements:

Mark. Represented by the **<mark>** tag. The mark element, as its name indicates, is for text marking purposes. It is used to highlight text (usually in yellow) that is meaningful to the author.

The following example shows the mark element.

Mark Element

This is **<mark>marked</mark>** text

Meter. Represented by the **<meter>** tag. This element is used to mark up measurements within a known range. Most of the time, this will be a scalar measurement, such as grades, rates, and so on. The **meter** element can make use of the following attributes that apply the values for the meter:

- **Value**—the current value shown by the meter
- **Min**—the minimum value
- **Max**—the maximum value
- **High**—the highest value the **meter** can be set to (can be less than or equal to the **max** attribute value)
- **Low**—the lowest value the **meter** can be set to (can be greater than or equal to the **min** attribute value)
- **Optimum**—the optimum value.

The following example shows how to use the meter element.

Meter Element

```
<meter value="70" min="0" max="100" low="54" high="100" optimum="100">70</meter>
```

Progress. Represented by the `<progress>` tag. This element represents the completion progress of a task, such as saving or downloading a file, which changes over time. The element has the following attributes:

- **Value**—the current value of the task progress
- **Max**—the maximum value of the task progress

The following example shows how to use the progress element.

Progress Element

```
Your download is <progress max="100" value="60">60%</progress> complete
```

Output. Represented by the `<output>` tag. This element is used to represent the result of a calculation or operation, such as the sum of two numbers that output a specific result.

The following example shows the use of the output element.

Output Element

```
Output: <input type="text" value="1" /> + <input type="text" value="3" /> =  
<output>4</output>
```

Time. Represented by the `<time>` tag. This element is used to represent a specific moment in time. It can be used for dates, times, or combinations of the two. It includes the **datetime** attribute, which is optional and not visible to the user.

The following example shows how to use the time element.

Time Element

```
The time is now <time datetime="2011-09-28T18:36">6:36 P.M. on November 28th</time>
```


Demonstration: Inline Semantic Elements

- Creating a simple webpage using the new HTML5 inline semantic elements

This demonstration shows how to:

- Create a new Empty Web Application in VS2010
- Add a new HTML file
- Add markup to the web page using the new HTML5 inline semantic elements

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file's body element add the following elements:

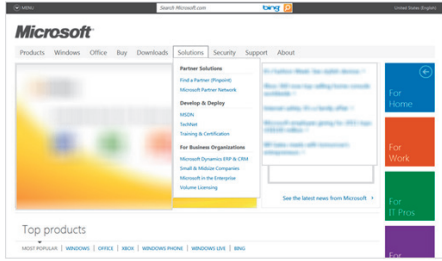
```
<body>
  This is<mark>marked</mark> text      <p>Your test score in the test was <meter
value="70" min="0" max="100" low="54" high="100" optimum="100">70</meter></p>
  <p>Downloaded: <progress value="16" max="48">33%</progress> of the content</p>
  <p>Output: <input type="text" value="1" /> + <input type="text" value="3" /> =
<output>4</output></p>
  <p>The time is now: <time>2011-11-12T14:54Z</time></p>
</body>
```

4. Save the file as an HTML file.
5. Run the saved file.

Interactive Elements

- Interactive elements include:
 - Details
 - Summary
 - Menu
 - Command

```
<details>
  <summary>The summary</summary>
  <p>Some details</p>
</details>
```



Interactive elements provide an interface for the user to interact with the webpage. The rationale of this group of elements is to replace some common implemented behaviors, such as commands and collapsible panels. In the previous module, you created a collapsible panel behavior. This behavior is so common that a new element, the **details** element, was created to replace the JavaScript collapsible panel code, presenting the same functionality. Other elements, such as **menu** and **command**, are also common behaviors that will, in HTML5, be implemented by the browser without any developer JavaScript code.

The following examples describe the interactive elements and how to use them:

Details. Represented by the **<details>** tag. This element is used for collapsible content that can be omitted without changing the meaning of the main content. When combined with the summary element, it will show the summary element when it is collapsed. The element has a main attribute: **open**. The **open** attribute can get a Boolean value that is used to indicate whether the details are collapsed or not. If the attribute does not exist, the details are collapsed and only the summary element is shown.

Summary. Represented by the **<summary>** tag. This element presents a summary, caption, or legend for the content found in its **details** parent element. When the **details** element is collapsed, this summary, caption, or legend will be shown instead.

The following code shows the details and summary elements.

Details and Summary Elements

```
<details>
  <summary>The summary</summary>
  <p>Some details</p>
</details>
```



Note: According to HTML5 specifications, a summary element is only used inside a details element. You cannot use it outside a details element.

Menu. Represented by the `<menu>` tag. This element presents a list of commands. It will be discussed more thoroughly in the next lesson.

Command. Represented by the `<command>` tag. It is used to represent a command that the user can invoke.

Other New Elements

- There are many more HTML5 elements, such as:
 - Video
 - Audio
 - Canvas
- These elements will be discussed later in the course



HTML5 introduces almost 30 new elements. Some of the most important ones, such as **video**, **audio**, and **canvas**, will be discussed later in the course.

 **Additional Reading:** You are encouraged to read the HTML5 specifications to learn about them at the following link: [HTML5 Specifications](#)

Lesson 2

Navigation and Menus

- Navigation Challenges
- The nav Element
- The menu Element

The two most fundamental goals of the World Wide Web are to make it easy to publish content and to link related content. HTML is used as the publishing format, and anchor (`<a>`) tags are used to create links. When users go anywhere on the web, they do so by following links, going from one piece of information to another. However, the web has grown tremendously since its beginning. The amount of content, and with it the number of links, has grown exponentially. Organizing all these links has become quite a challenge.

The use of HTML is no longer reserved for traditional websites alone. HTML is also used to build robust and highly interactive web applications that look and behave very much like desktop applications. More and more frameworks make use of HTML in desktop applications, such as Windows 8 Metro-style applications.

Such applications have often used links and buttons interchangeably, similar to how toolbars and context menus work in regular desktop applications. However, there has been no standard or consensus on how to create these menus in web applications, no common mechanism for interacting directly with the browser, and no way to distinguish between links used to connect related information and links used as actions or commands.

This lesson introduces the **nav**, **menu**, and **command** elements, which were added to HTML5 to help resolve these issues.

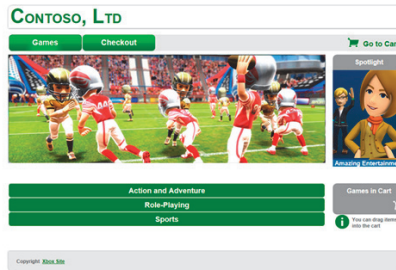
Lesson Objectives

After completing this lesson, you will be able to:

- Describe the challenges in creating website navigation.
- Create webpage navigation.
- Use menus.

Navigation Challenges

- Anchor tags all look the same
- Navigation systems have been mostly visual:
 - Relying on visual information hurts accessibility
 - Search engines cannot interpret visual information
- Differentiating between links and commands can be difficult



Anchors do not differentiate

When an anchor element contains an **href** attribute with a value, a link is created. The link may point to any type of content, including (but not limited to) the following items:

- A webpage on the current website
- A webpage on another website
- A named anchor on the current webpage
- An email address
- A resource that uses a different protocol (for example, FTP)
- A JavaScript function

While the anchor tag is fundamental to the web, there is no way for a visitor to tell, unassisted, where a link will lead to, or what it will do. The link text is usually brief, often a word or two, and almost never tells the visitor what will be the result of clicking the link.

The following examples all look the same to the user, but behave very differently from one another.

Different Types of Links

```
<a href="/ContactForm.aspx">Contact Us</a><!-- will navigate to another page on the same
site, presumably with a form that the user can fill out -->
<a href="#" onclick="submitForm()">Contact Us</a><!-- will call a JavaScript function,
perhaps to submit the form asynchronously -->
<a href="mailto:webmaster@example.com">Contact Us</a><!-- will instruct the browser to
open the default email program on the user's computer and to create a pre-addressed email
message to webmaster@example.com -->
```

Previous navigation systems are mostly visual

Web developers use several techniques to help users differentiate the various types of links. The most common approaches use one or more of the following techniques, though there are quite a few more:

- Links to other pages in the same site are generally grouped into one or two navigation areas that are visible on most of the website pages.
- The navigation areas generally have some visual element, such as a different background color, that differentiates them from the rest of the site.
- Links to other sites may appear on a separate page set aside especially for that purpose.
- Links to other sites may appear in a section specifically devoted to external links.
- Links to other sites may appear with an icon indicating that the link opens a separate site.

The problem with all of the outlined techniques is that they rely very heavily, even exclusively, on visual cues, such as icons, background colors, and different styles. Even if a page is set aside for external links, the standard navigation areas with links to other pages on the same site are usually also present.

Relying on visual information hurts accessibility

While these visual cues may not be a problem for most visitors, they alienate everyone else. Visitors who are blind or have low vision, for example, use special screen-reader programs, and can have great difficulty navigating sites that rely on visual cues alone. In addition, styles and layout are often defined (as they should be) in separate CSS files, which screen-readers cannot interpret. They rely on the document structure they see in the HTML markup, and on contextual information. They use additional algorithms and techniques to try to figure out where the different parts of the page are, to make it easier for their users to quickly find the content they are looking for.

People who are blind or have low vision are not the only ones who are alienated. Smartphone users, too, can have difficulty, because smartphones have smaller screens that can display less information at a time. When pages are laid out visually, they often impose constraints appropriate for larger desktop monitors, with minimum widths that are too wide for smartphone screens. While some smartphone browsers can ignore some of these constraints, they still have difficulty focusing on the relevant content or identifying the navigation areas, which could help improve the browsing experience.

Search engines cannot interpret visual information

Search engines scour the web, collecting, indexing, and analyzing links and content. Search engines rely very heavily on links to determine whether a given webpage might answer a particular search. When a search engine visits a website, it analyzes all of its links and tries to determine which links are more important than others. Search engines find it difficult to reliably distinguish between links in the navigation areas and other links on the page, because, like screen readers, they cannot interpret visual information.



Note: Search engines do have some techniques that help them identify navigation areas in some cases. For example, research done by various search engines and other organizations has shown that there are elements that repeat on a great number of websites. One example is the use of a CSS class name such as **.nav** or **.menu** to associate a navigation area with a particular set of styles in the site's CSS files. Relying on these techniques has proven reliable in many cases, but good results are not predictable or guaranteed.

The ability of search engines to distinguish between different kinds of links is important not only for the users, but for the site also. Most people use search engines these days to find the content they want. When a search engine determines that your site has that content, it can bring users to your site and help grow your business. Web developers go to great effort to let search engines know what kind of content can be found on their sites. They even submit *sitemaps*, files that extensively delineate the structure of the

site and where important pages can be found, directly to the search engines. However, sitemaps are not necessarily dynamic; they do not automatically change when pages are added, removed, or simply modified for example, when their titles and content change).

Differentiating between links and commands can be difficult

When websites perform like desktop applications, it can be difficult for both search engines and screen readers to tell the difference between `<a>` elements that navigate and `<a>` elements that invoke actions. For example, a site may feature a text editor with commands that support making selected text bold, italic, or underline. Each of these commands may be represented in the HTML document with an `<a>` element, which can be indistinguishable from a navigational link.

The following links run JavaScript functions and may not be easy to distinguish from navigation links.

Example HTML showing a possible text editor implementation

```
<div id="textEditor">
<ul>
<li><a href="#" onclick="toggleBold();">B</a></li>
<li><a href="#" onclick="toggleItalic();">I</a></li>
<li><a href="#" onclick="toggleUnderline();">U</a></li>
</ul>
<textarea></textarea>
</div>
```

To apply bold, italic, or underline styles to the text, JavaScript must be used. In the preceding example, the **onclick** attribute of each anchor element calls the relevant JavaScript function when the element is clicked. These anchor elements are used as commands, rather than as links.



Note: Calling JavaScript functions inline is strongly discouraged, as explained in Module 1. The preceding example violates that convention only to make a point about the anchor elements. A real application should wire up the JavaScript functions by using JavaScript *in an external file*.

It can be challenging to tell commands and links apart because they share the same syntax and are used interchangeably. An external tool or service, such as a screen reader or a search engine, can guess at the meaning of any given anchor element, but it can never be completely certain that it knows what the web developer intends. HTML5 adds new semantic elements to facilitate this communication; the web developer can use the new elements to tell the browser, the search engine, and the screen reader what role each link plays.

The nav Element

The **nav** element:

- is a semantic element
- is a container for a navigation area
- is designed to help search engines and screen readers
- Is used for:
 - *document navigation*, when placed inside an **article** element
 - *site navigation*, anywhere else

```
<nav>
  <ul>
    <li><a href="/">Home</a></li>
    <li><a href="/products">Products</a></li>
    <li><a href="/about">Meet the Team</a></li>
  </ul>
</nav>
```

Overview

HTML5 adds a new **nav** element as a container used to mark areas of the webpage that contain navigation. This element reduces the reliance on visual information by explicitly stating where the navigation areas are in the markup. This is why the **nav** element is referred to as a semantic element; it assigns a very specific meaning ("navigation area") to whatever group of links it is wrapped around.

Syntax

The **nav** tag has a very simple syntax. It is a simple tag that uses no special attributes. It supports all the global attributes, including **id** and **class**, although these usually are not needed. The **nav** element typically wraps other containers that contain anchor elements.

If the main navigation menu for the site is an unordered list of links, the nav element would simply wrap it as shown in the following code.

The nav Element as a Container

```
<nav>
  <ul>
    <li><a href="/">Home</a></li>
    <li><a href="/products">Products</a></li>
    <li><a href="/about">Meet the Team</a></li>
  </ul>
</nav>
```

The preceding example shows some standard navigation menu markups wrapped with the **<nav>** tag. Before HTML5, it would have been difficult for screen readers and search engines to identify it as a navigation area. The new **<nav>** tag is added to make this clear.



Best Practice: The **nav** element can contain many different types of content. It can contain paragraphs that describe the target of each link prosaically, links inside **div** or **span** elements, and so on. However, unless there is a special reason to use another structure, unordered lists (the

ul element) are strongly recommended. Semantically, such lists inside **nav** elements describe the most common intent: an unordered (that is, *unnumbered*) list of links that constitute a navigation area.

Usage

The **nav** element can be used for two different purposes: site navigation and document navigation. It is often recommended that the **nav** element be placed inside the **header** element, because this most clearly expresses the intent of the navigation area, but this is not a strict requirement. Website design allows for great variation, so the **nav** element may be placed in the **footer**, in a sidebar (although probably not in an **aside** element), or even as a direct child of the **body** element.

Anchors should not be direct children of the **nav** element.

Site Navigation

While the **nav** element can contain links to external sites, it is primarily meant for links to other webpages within a single site. The main navigation menus on most sites are a good example of this kind of use, as demonstrated in the preceding example.

Because navigation menus generally serve as a good outline for the structure of a website, search engines benefit because they can easily determine the role of the links inside the navigation areas. Screen readers, too, can quickly skip over the navigation area when the user asks for the page content, or focus in on the navigation areas when the user wants to go somewhere else on the site.

Document Navigation

The other use of the **nav** element is for navigation inside documents, such as magazine and newspaper articles, encyclopedia entries, books, and help sections. Such documents are often split into smaller sections, such as chapters, prefaces, and references. When named anchors are used for each section, the **nav** element can contain a list of links to all those sections, and therefore, can serve as a table of contents. These sections may be located on the same page or spread across multiple pages; in the latter case, the anchors would link to the other pages, and possibly to named anchors within those pages.

The difference between the two uses is very simple. When the **nav** element is placed inside an **article** element (a semantic element, described in the preceding lesson that marks its content as a document), it represents a table of contents for that article, and can be identified as such by screen readers and search engines. In any other place, the **nav** element is used for site navigation.

The body of a webpage with a news article may look something like the following example.

Example of two nav elements being used for site and document navigation

```
<body>
<header>
<h1>Something Important Happened Somewhere</h1>
<nav><!-- Site Navigation -->
<ul>
<li><a href="/">Home</a></li>
<li><a href="/entertainment">Entertainment</a></li>
<li><a href="/politics">Politics</a></li>
<li><a href="/sports">Sports</a></li>
</ul>
</nav>
</header>
<article>
<header>
<p>Table of Contents</p>
<nav>
```

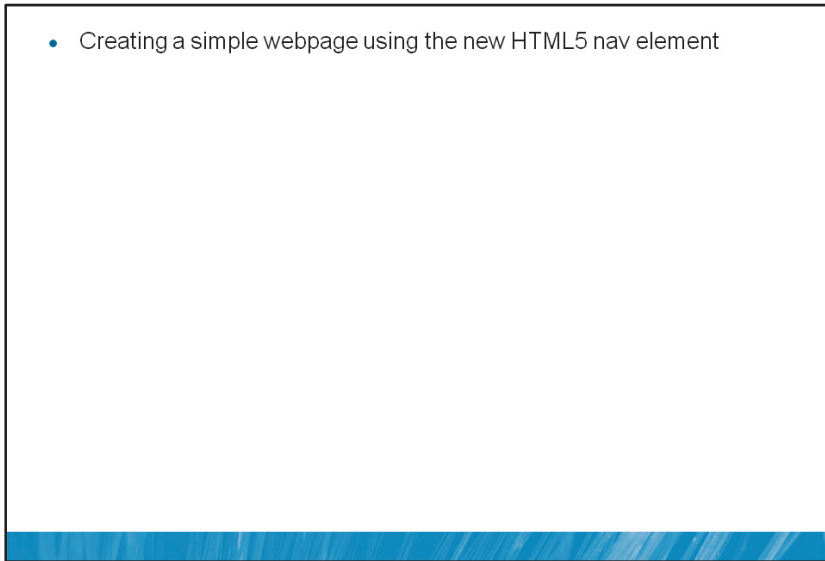
```

<ul>
<li><a href="#summary">Summary</a></li>
<li><a href="#details">The Details</a></li>
<li><a href="#sources">Sources</a></li>
</ul>
</nav>
</header>
<section>
<a id="summary" />
<header>
<h2>Somebody did Something Vital Yesterday</h2>
</header>
    This is the summary of the article.
</section>
<section>
<a id="details" />
    The main content of the article would be here.
</section>
<section>
<a id="sources">
<header>
<h2>Sources</h2>
</header>
<ul><!-- a link menu that is not wrapped in a nav element -->
<li><a href="http://www.example.com/link1">A Well Known Source</a></li>
</ul>
</section>
</article>
</body>

```

The sample HTML contains two **nav** elements. The first **nav** element wraps a list of links to other pages on the same site and is used for site navigation. The second **nav** element wraps a list of links to sections of the article, and represents the table of contents. In this case, all the links in the table of contents are links to other sections of the same page; however, if the article was split up over multiple pages, links to sections of the article appearing on other pages would still appear in that same table of contents.

Demonstration: The nav Element



This demonstration shows how to:

- Create a new Empty Web Application in VS2010
- Add a new HTML file
- Add markup to the web page using the new HTML5 nav element

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file's body element add the following elements:

```
<body>
  <header>
    <h1>
      Something Important Happened Somewhere</h1>
    <nav>
      <!-- Site Navigation -->
      <ul>
        <li><a href="/">Home</a></li>
        <li><a href="/entertainment">Entertainment</a></li>
        <li><a href="/politics">Politics</a></li>
        <li><a href="/sports">Sports</a></li>
      </ul>
    </nav>
  </header>
  <article>
    <header>
      <p>
        Table of Contents</p>
      <nav>
        <ul>
          <li><a href="#summary">Summary</a></li>
          <li><a href="#details">The Details</a></li>
          <li><a href="#sources">Sources</a></li>
        </ul>
      </nav>
    </header>
  </article>
</body>
```

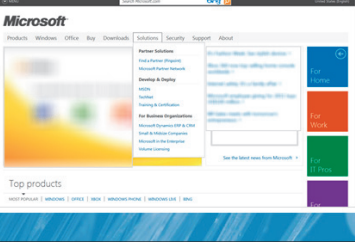
```
        </ul>
      </nav>
    </header>
    <section>
      <header>
        <h2>
          Somebody did Something Vital Yesterday</h2>
        </header>
        <p>
          This is the summary of the article.</p>
        </section>
        <section>
          The main content of the article would be here.
        </section>
        <section>
          <header>
            <h2>
              Sources</h2>
            </header>
            <ul>
              <!-- a link menu that is not wrapped in a nav element -->
              <li><a href="http://www.example.com/link1">A Well Known
Source</a></li>
            </ul>
          </section>
        </article>
      </body>
```

4. Save the file as an HTML file.
5. Run the saved file.

The menu Element

- The **menu** element:
 - contains commands
 - can be a toolbar or a content menu
 - is designed for browser integration
 - can be nested
- Almost any element can be used as a command

```
<menu>
  <button id="searchButton">Search</button>
  <button id="helpButton">Help</button>
</menu>
```



Overview

Modern web applications often have highly interactive clients. In other words, they tend to have a lot of JavaScript code, they respond to user actions, and they let users run many commands, which can be found on toolbars and context menus. The new **menu** element introduced in HTML5 is a semantic element that is used to represent toolbars, context menus, and other lists of commands.



Note: The **nav** element and the **menu** element are sometimes confused with each other, but they serve very different purposes. The following table shows some of the more important differences between them.

	nav	menu
What is it used for?	Navigation	User interaction
Which websites use it?	Almost every website	Highly interactive web applications
Contents	Links	Commands
Dependencies	Almost none	JavaScript

Advantages of the menu Element

The **menu** element is designed to solve a number of problems:

- First, prior to HTML5, there was no consensus or standard that governed the use of menus and commands in highly interactive web applications. This hurt the user experience, because users had to learn a new paradigm with each application. The **menu** element standardizes the mechanism for

exposing commands to the user so that the user can interact with every application in the same basic way.

- The **menu** element provides stronger interaction with the web browser than was previously possible. Because the **menu** element is a semantic element, the browser can “understand” the element and build command menus on the browser shell, outside the rendered document. This makes commands more visible and lets web developers build web applications that look and feel even more like standard desktop applications.
- The **menu** element clearly designates the **a** elements it contains as commands; these links were previously ambiguous to search engines and screen readers, but can now be safely ignored.

Menu Types

The **menu** element uses a **type** attribute that can have one of three values: **toolbar**, **context**, or **list** (the default value). The **menu** element can behave just like an unordered list (a **ul** element), so its children can be list items (**li** elements). Each list item should contain a command, such as a **command** or a **button** element. Commands can also be added directly as children of the **menu** element.

The following example demonstrates adding buttons to the menu element.

Buttons as Direct Children of the menu Element

```
<menu>
<button id="searchButton">Search</button>
<button id="helpButton">Help</button>
</menu>
```

Commands and the **button** element will be explained in the next topic.

Toolbars

When the **type** attribute is set to **toolbar**, the menu represents a toolbar. A toolbar menu is a menu that is only visible on the page when it is needed. The advantage of using a toolbar menu is that supporting browsers understand it and can display the commands in a special menu in the browser window. Not all browsers are required or expected to support this behavior. The menu element is designed to work even without browser shell support. The contents of a toolbar menu may change as needed: menu items can be removed, added, disabled, or enabled, as the context changes.

The following example demonstrates a simple toolbar menu.

A Simple Toolbar Menu

```
<menu type="toolbar">
<li><button type="button" title="Search" id="searchButton">Search</a></li>
<li><button type="button" title="Help" id="helpButton">Help</a></li>
</menu>
```

Context Menus

When the **type** attribute is set to **context**, the menu is a context menu: a menu that appears, for instance, when the user right-clicks a relevant section. Supporting browsers add the commands of the context **menu** element to the default context menu of the browser when it is opened. Different browsers may support different behaviors, but they will generally merge their built-in context menus with the element commands. A context menu can be defined as shown in the toolbar example, except for the change in the **type** attribute.

List Menus

When the **type** attribute is omitted or set to **list**, it is a general menu. It can be used anywhere, and it does not use any special browser integration.

The following example shows a menu element that contains three commands enclosed in `li` elements.

A menu element with the list type

```
<menu type="list">
<li><button type="button" title="Bold" id="toggleBoldButton">B</button></li>
<li><button type="button" title="Italic" id="toggleItalicButton">I</button></li>
<li><button type="button" title="Underline" id="toggleUnderlineButton">U</button></li>
</menu>
```

Nested Menus

Menus can also be nested. The outer menu will generally be a toolbar or context menu, and the inner menus will always be list menus. When the browser builds toolbar or context menus, it will also build submenus for the nested menu elements. To show a caption for the nested menu, use the **label** attribute, or a **label** element that points to the **menu** element with its **for** attribute.

The following example demonstrates a nested toolbar menu.

A Nested Toolbar Menu

```
<menu type="toolbar">
<li><button type="button" title="Search" id="searchButton">Search</a></li>
<li>
<menu label="Edit">
<li><button type="button" id="toggleBoldButton">Bold</button></li>
<li><button type="button" id="toggleItalicButton">Italic</button></li>
<li><button type="button" id="toggleUnderlineButton">Underline</button></li>
</menu>
</li>
<li><button type="button" title="Help" id="helpButton">Help</a></li>
</menu>
```

Commands

The command Element

HTML5 introduces a new **command** element. The **command** element has two separate uses: one for its appearance in a **menu** element, and the other anywhere else. When a **command** element is placed in a **menu** element, it is a very generic kind of command that can be used to invoke some action. When a **command** element is placed elsewhere, it is not rendered visually and can be used to associate some behavior with a keyboard shortcut.

When a **command** element is placed inside a **menu** element, its **label** attribute must be set. The **label** attribute is used as the command caption. To set the kind of behavior the **command** element should implement, use the **type** attribute. When the **type** is set to "command" or is omitted ("command" is the default value), the **command** element is used to invoke an action.

The following example shows a simple command element, with the label "Search," that invokes a JavaScript function.

A Simple Command Element in a Menu

```
<menu>
<command type="command" label="Search" onclick="search()" />
```



```
</menu>
```

To use the **command** element as a radio button, set the **type** attribute to "radio". To associate several radio commands with each other, set the **radiogroup** attribute for each of them to the same value. To select a specific radio command from a group of radio commands, set its **checked** attribute to "checked". The **checked** attribute is a Boolean attribute.

The following example presents a group of radio commands.

A Group of Radio Commands

```
<menu>
<command type="radio" radiogroup="priority" label="High Priority"
onclick="togglePriority('high')" />
<command type="radio" radiogroup="priority" label="Medium Priority"
onclick="togglePriority('medium')" checked="checked" />
<command type="radio" radiogroup="priority" label="Low Priority"
onclick="togglePriority('low')" />
</menu>
```

To use the **command** element as a check box, set its **type** attribute to "checkbox". To select the check box command, set its **checked** attribute to "checked". Remove the **checked** attribute to deselect the command.

The following example presents two check box commands.

Check box Commands

```
<menu>
<command type="checkbox" label="Bold" onclick="toggleBold()" />
<command type="checkbox" label="Italic" onclick="toggleItalic()" checked="checked" />
</menu>
```

Types of Commands

Besides the **command** element, any element that performs some action is considered a command and can be added to a menu. Every command must abide by two constraints:

- Each command must have a caption of some sort. This can be a **label** attribute, a **label** element, or textual content.
- Each command must define some action.

Every command that abides by these two constraints can be added to a **menu** element. The following list describes the many types of commands supported in HTML5:

- The new **command** element is designed to be used with the **menu** element.
- The **a** element can be used as a command if it abides by the constraints. Its defined action can be a JavaScript function in the **onclick** event or an **href** URL value. An invisible, named anchor with no link or event handler cannot serve as a command.
- The **button** element is a richer alternative to the **<input type="button">** element. It can be used anywhere an input button can be used. Unlike the input button, it can show stylized and multi-line text.
- An **input** element whose type is any kind of button (including submit, reset, radio, and check box buttons) or a check box can also be a command. In the case of the radio and check box commands, any menu items built by the browser should reflect the checked or selected state of the command.

- An **option** element inside a **select** element behaves like a radio or check box command, depending on whether the **multiple** attribute of the **select** element is set.
- Any element that has its **accesskey** attribute set can become a command. This provides tremendous flexibility; simply adding keyboard support to an element can make it usable in a menu.

Lab: Creating Page Structure and Navigation

- Using Semantic Elements in the Home Page
- Creating the Game Details and Cart Pages
- Adding Navigation and Menus to the Home Page

Logon Information

Virtual Machine	Virtual machine: SEA-DEV
User Name	User name: Admin
Password	Password: Pa\$\$w0rd

Estimated Time: 45 minutes

Scenario

After creating the home page by using basic HTML knowledge, management asks that you incorporate the new semantic and structural elements of HTML5. There is also a requirement for site navigation. The navigation must connect all the pages that will be built in the future. You will also add two more pages to the web application: the Game Details page and the Cart page.

Objectives

After completing this lab you will:

- Integrate the new HTML5 elements into the home page.
- Create the game details and cart pages.
- Create a navigation element for the web application.

Exercise 1: Using Semantic Elements in the Home Page

Scenario

In this lab, you will start with the HTML output of the lab from Module 1 and migrate it to HTML5 by using the new structural and semantic elements discussed in this module.

► **Task 1: Replace the existing HTML elements with the new HTML5 elements.**

1. Open the starter solution, which is located in the **D:\Mod02\Labfiles\Starter** folder.
2. Go over the **homepage** and replace the existing HTML elements (mostly **divs**) with the more meaningful HTML5 elements.

► **Task 2: Update the style to reflect the HTML changes.**

1. Go over the CSS files and update the style to reflect the changes that were made to the homepage HTML.

Results: A home page that is written with HTML5 semantic elements.

Exercise 2: Creating the Game Details and Cart Pages

Scenario

In this lab, you will create two additional webpages by using the HTML5 semantic and structural elements that you learned about in the module. The completed webpages will be used in the last exercise to enable navigation inside the web application.

► Task 1: Create the Game Details webpage.

1. Create the **GameDetails.html** file.
2. Create the game details markup by using the provided design image, **GameDetails.png**, which is located in **D:\Mod02\Labfiles\Starter\Design**.

► Task 2: Create the Cart webpage.

1. Create the **Cart.html** file.
2. Create the cart markup by using the provided design image, **Cart.png**, which is located in **D:\LabFiles\Starter\M02\Design**.

► Task 3: Add CSS styles to the Game Details webpage.

1. Add the **game.css** file.

Results: Full game details and cart pages are written with HTML5 semantic elements.

Exercise 3: Adding Navigation and Menus to the Home Page

Scenario

In this lab, you will connect the homepage to the Cart and Game Details webpages.

► Task 1: Add links from the Homepage to the other webpages.

1. Add links in the **homepage.html** that point to the other webpages.

Results: Home, Game Details, and Cart pages that incorporate navigation and menus are created.

Module Review and Takeaways

In the module, you learned about the new HTML5 elements. You learned the rationale behind the creation of these elements, and how and when to use them. You also learned the use of navigation and menus, without which the World Wide Web would not be as easy to browse as it is today.

Review Questions

Why should developers migrate from HTML4 elements to HTML5 elements?

Real-World Issue and Scenario

The migration from HTML4 to HTML5 should not be done without thinking. This is a very complicated task that may break the appearance of the webpage if not done carefully.

Module 3

Creating Form Input and Validation

Contents:

Module Overview	3-2
Lesson 1: Working with Input Types	3-3
Lesson 2: Using Form Attributes	3-16
Lesson 3: Validation	3-35
Lesson 4: Using Browser Detection, Feature Detection, and Modernizr	3-44
Lab: Creating Form Input and Validation	3-55
Module Review and Takeaways	3-59

Module Overview

- Working with Input Types
- Using Form Attributes
- Validation
- Using Browser Detection, Feature Detection, and Modernizr

In this module, you will review the creation of web forms by using the new input types and attributes in HTML5. You will also learn about form validation.

In addition, you will learn about browser detection, feature detection, and the Modernizr JavaScript library, which helps detect HTML5 features. Many new HTML5 features are not yet supported by all the major browsers. When you develop websites and applications, you must know which features exist so you can implement fallback alternatives for missing features.

Objectives

After completing this module, you will be able to:

- Use the new input types.
- Apply the new attributes.
- Integrate validation into webpages.
- Use browser detection, feature detection and Modernizr.

Lesson 1

Working with Input Types

- Receiving Input by Using Input Types
- Date Pickers
- New Text Box Types
- General Input Types

HTML5 introduces a number of new input types. These input types were created because their element behaviors are commonly used in websites and applications. The new input types shift the burden of responsibility for implementing these behaviors from the developer to the browser. For example, a text input type can now be replaced with an email input type or URL input type that is validated by the browser, freeing the web developer from having to write a custom validation script. Inputs types requiring a value within a range will be depicted by the browser as a visible slider, freeing the web developer from having to create a JavaScript plug-in that implements this kind of UI.

The new input types therefore give developers more spare time to work on creating other behaviors in their websites and applications. Also, the need for external plug-ins such as a number or color picker might decrease.

This does not mean that developers will never write JavaScript behaviors. The HTML5 specifications do not provide solutions for every possible use of these input types, but rather fulfill common requirements. It is the responsibility of the developer to meet the unique requirements of the website or application.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the new HTML5 input types.
- Apply date pickers on webpages.
- Use the new text box types on webpages.
- Use the new interactive text box types on webpages.

Receiving Input by Using Input Types

- A lot of common web behaviors were missing
 - Email and URL validation
 - Date pickers
 - Sliders
- New form elements introduced in HTML5

Before HTML5, when a web developer wrote a form to receive information from the user, he or she had very limited options. Even though a developer could use text boxes, check boxes, radio buttons, and drop-down lists to create a solid input form, they still frequently had to add other input behaviors such as URL and email validation, and date selection. Web developers were forced to use external JavaScript libraries (such as jQuery UI) or widgets, or write their own behaviors by using JavaScript.

In writing the new HTML5 specifications, W3C recognized this deficiency and introduced 12 new input types to fill the need. The new input types offer a variety of behaviors, and help create forms more easily.

It should be noted that support for the new input types is only being implemented in new browsers such as Internet Explorer 10. Older browsers gracefully downgrade the new types into text boxes if they are not supported. In these cases, using feature detection (which will be described in Lesson 4) can help replace the needed behaviors, if the browser does not support them.

Date Pickers

- Date pickers are used on many different web forms
- In HTML5, date pickers are rendered by the browsers
 - Minimize the need for widgets or JavaScript UI libraries
- Include the following types: **date**, **month**, **week**, **time**, **datetime**, and **datetime-local**

```
<input type="date" name="dpBirthDate" value="2011-12-19" />
```

2011-12-19

Date pickers are widely used in websites and applications. Most web forms use them to receive as input a date or time. However, even though date pickers are so frequently needed, they had not been included in any HTML specifications before HTML5. Instead, web developers used external JavaScript libraries or built their own date pickers. This of course was a cumbersome task, because writing a date picker is not easy. Date pickers must support internationalization into the many languages of the world. Depending on the context, date pickers must support picking specific dates, picking specific times, and more. Date pickers must also support time zones and Coordinated Universal Time (UTC).

HTML5 introduces a new set of date picker input types, making the browser responsible for creating and implementing its own UI and behavior. The HTML5 specifications provide details of how the behavior should be implemented by the browser vendors. This helps developers who use the date pickers to understand what the expected behavior should be. If the behavior does not fit the needs of the developer, he or she can still use external JavaScript libraries and widgets.

The following examples show the new date pickers:



Note: All the date pickers in the example have the same naming convention. The date pickers use a name attribute that start with dp to indicate date picker.

date. An input type that collects a specific date without an associated time. It is used when the form requires a date (such as a birth date), and time is not relevant.

The following code shows how you can use the date input type to request a birth date.

The date input type

```
<input type="date" name="dpBirthDate" value="2011-12-19" />
```

month. An input type for collecting a month and its associated year. This input type is often used to collect data about credit card expiration dates, or in other circumstances in which only the month is needed.

The following code shows how you can use the month input type to request a credit card expiration.

The month input type

```
<input type="month" name="dpCCEExpiration" value="2011-12" />
```

week. An input type for collecting a specific week and its associated year. This type is commonly used in Europe.

The following code shows how you can use the week input type to request an arrival week.

The week input type

```
<input type="week" name="dpArrivalWeek" value="2011-W16" />
```

time. An input type for collecting a specific time without an associated date. This type is used in situations where the time is important, but the date is either not necessary or already known, such as the desired departure time of a flight.

The following code example shows how you can use the time input type to request a departure time.

The time input type

```
<input type="time" name="dpDepartureTime" value="17:39:57" />
```

datetime. An input type for collecting a specific date and a specific time on that date, including time zone information. This is probably the most common type.

The following code example shows how you can use the datetime input type to request a birth date.

The datetime input type

```
<input type="datetime" name="dpBirthDate" value="2011-12-31T23:59:60Z" />
```

datetime-local. An input type for collecting a local date and time. In contrast to the datetime type, this type does not collect time zone information.

The following code example shows how you can use the datetime-local input type to request a birth date.

The datetime-local input type

```
<input type="datetime-local" name="dpBirthDate" value="2011-12-19T16:39:57" />
```

The date picker input types can use attributes that help to create a more specific behavior such as:

- Step.** The step size used in the date picker. By default, it is set to 1. In every date picker, this attributes meaning is different. For example, in month, the step will be in months; in week, it will be in weeks, and so on.

Min. The minimum value of the date picker.

Max. The maximum value of the date picker.

Value. The default value of the date picker. The value of this attribute will be replaced by the value selected by the user.



Note: The appearance of the date picker depends on the browser. The specifications only describe the behavior of the input type, and not appearance.

The new date pickers decrease the need for external JavaScript libraries and allow the web developer to rely on the native support of the browser for the needed behavior.

Demonstration: Date Pickers

- Creating a simple webpage using the new HTML5 date picker elements

This demonstration shows how to:

- Create a new Empty Web Application in VS2010
- Add a new HTML file
- Add markup to the web page using the new HTML5 date picker elements

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file's body element add the following elements:

```
<body>
  <form>
    <label for="dpBirthDate">Date: </label><input type="date" name="dpBirthDate"
value="2011-12-19" /><br/>
    <label for="dpCCExpiration">Month: </label><input type="month"
name="dpCCExpiration" value="2011-12" /><br/>
    <label for="dpArrivalWeek">Week: </label><input type="week" name="dpArrivalWeek"
value="2011-W16" /><br/>
    <label for="dpDepartureTime">Time: </label><input type="time"
name="dpDepartureTime" value="17:39:57" /><br/>
    <label for="dpBirthDate">Datetime: </label><input type="datetime"
name="dpBirthDate" value="2011-12-31T23:59:60Z" /><br/>
    <label for="dpBirthDate">Datetime-local: </label><input type="datetime-local"
name="dpBirthDate" value="2011-12-19T16:39:57" /><br/>
  </form>
</body>
```

4. Save the file as an HTML file.
5. Run the saved file.

New Text Box Types

- New text box types that can perform input validation:
 - email
 - url
- New **types with no specific behavior**:
 - tel
 - search

```
<input type="email" name="txtEmail"
value="email@company.com" />
<input type="url" name="txtUrl"
value="http://www.microsoft.com" />
<input type="tel" name="txtPhoneNumber" />
<input type="search" name="txtSearch" />
```

HTML5 adds several new text box input types for common types of data, such as email addresses, URLs, telephone numbers, and search boxes. For email entry and URL entry, the browser natively implements a validation of the inserted text, based on the specifications. The search box does not implement any special behavior, but allows browsers to differentiate it from regular text boxes. Some of the browsers render a magnifying glass icon in or near the search box to mark them as search boxes.

The new text box input types can be used in the following scenarios:

email. Used to indicate that the input text must be in the format of an email address. It validates the input to ensure that the text is indeed a valid email address. If the text is not in the format of a valid email address (for example, it does not include the @ symbol), most browsers will render a balloon with an error message or paint the border of the text box red. One of the attributes that can be used with this type is the multiple attribute. The multiple attribute can get a Boolean value to indicate whether the input types can have multiple emails as the input type value. If the attribute value is "true", the user can insert multiple email addresses separated with commas.

The following example shows the use of the email input type.

The email input type

```
<input type="email" name="txtEmail" value="email@company.com" />
```



Note: The email input type does not have to accept only a single email address; it can also accept as input a set of comma-separated strings, of which each one is a valid email address.

url. Used to indicate that the input text must be in the format of a URL. It validates the input to ensure that the text is indeed a valid URL. If the text is not a valid URL, most browsers will render a balloon with an error message or paint the border of the text box red.

The following example shows how to use the url input type.

The url input type

```
<input type="url" name="txtUrl" value="http://www.microsoft.com" />
```



Note: An empty string ("") and a URL surrounded by spaces are also valid inputs.

tel. Used to indicate that the input text is a telephone number. The input type has no specific behavior. The following example shows how to use the tel input type.

The tel input type

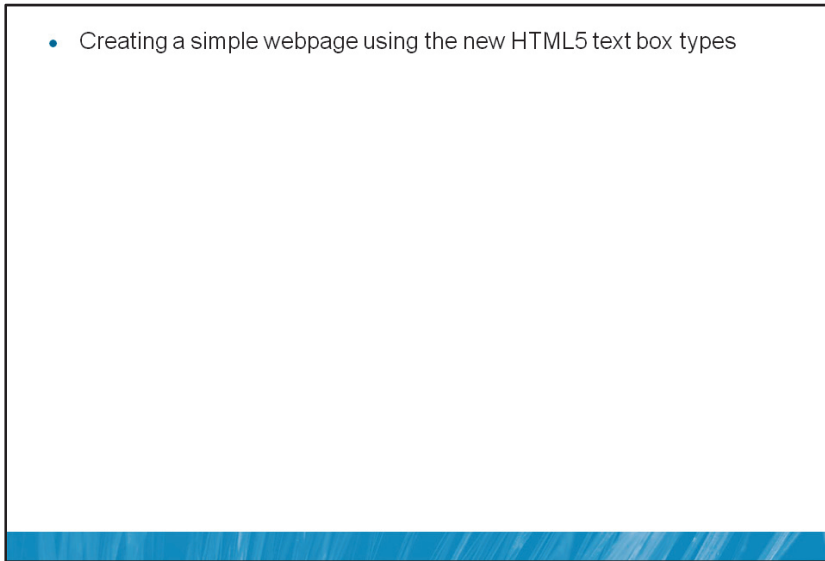
```
<input type="tel" name="txtPhoneNumber" />
```

search. Used to indicate a search box inside a form. The input type has no specific behavior. The following example shows how to use the search input type.

The search input type

```
<input type="search" name="txtSearch" />
```

Demonstration: New Text Box Types



This demonstration shows how to:

- Add markup to the web page using the new HTML5 text box types.

Demonstration Steps

1. In the HTML file's body element after the last `</br>` element, add the following elements:

```
<label for="txtEmail">Email: </label><input type="email" name="txtEmail"
value="email@company.com" /><br/>
<label for="txtUrl">Url: </label><input type="url" name="txtUrl"
value="http://www.microsoft.com" /><br/>
<label for="txtPhoneNumber">Telephone: </label><input type="tel"
name="txtPhoneNumber" /><br/>
<label for="txtSearch">Search: </label><input type="search" name="txtSearch" /><br/>
```

2. Save the file as an HTML file.
3. Run the saved file.

General Input Types

New types that interact with the user:

- **number**
- **range**
- **color**

```
<input type="number" name="txtAge" min="0" max="120" />
<input type="range" name="txtAge" min="0" max="120" />
<input type="color" name="txtCarColor" />
```

A web developer can use input types to gather and validate information from a user. HTML5 includes general input types such as a number picker, a range slider, and a color picker.

number. Used for input types that contain only numbers. Most browsers will render this type as a text box with two small arrow buttons pointing up and down. Pressing these buttons raises or lowers the value in the text box by the amount given in the step attribute. Another option to insert a value is by typing a valid floating point number into the text box. The main attributes of the number input type are:

max. The maximum value that can be used as an input.

min. The minimum value that can be used as an input.



Note: The specification does not include any behavior regarding what happens if a user types in a number greater than the max attribute or less than the min attribute. Most browser implementations will not allow the user to insert invalid numbers.

step. The amount that is added or subtracted from the current value when the up or down button is clicked. The step attribute can get a positive floating-point number as its value.

value. The current value in the text box. Also used to insert a default value when the page is first rendered.

The following example shows the use of a number input type.

The number input type

```
<input type="number" name="txtAge" min="0" max="120" />
```

range. Very similar to number. Used for input types that collect a value from a range of numbers. The main difference between number and range is their rendering: a range is rendered as a slider bar, while a number is rendered as a text box. The slider is a UI element that helps choose the relevant range by the user and will hold a number in the value attribute in the end. Both number and range have the same attributes.

The following example shows how to use the range input type.

The range input type

```
<input type="range" name="txtAge" min="0" max="120" />
```

color. Creates a text box that collects a string to represent a color (some browsers will render it as a color picker instead). The value attribute of the color must be made up of a # character followed by a 6-digit hexadecimal number, whose characters must be in the ranges 0–9, a–f, and A–F.



Note: Color keywords such as "red" or "blue" are not acceptable inputs.

The following example shows how to use the color input type.

Color input type

```
<input type="color" name="txtCarColor" />
```

Demonstration: General Input Types

- Creating a simple webpage using the new HTML5 general input types

This demonstration shows how to:

- Add markup to the web page using the new HTML5 general input types

Demonstration Steps

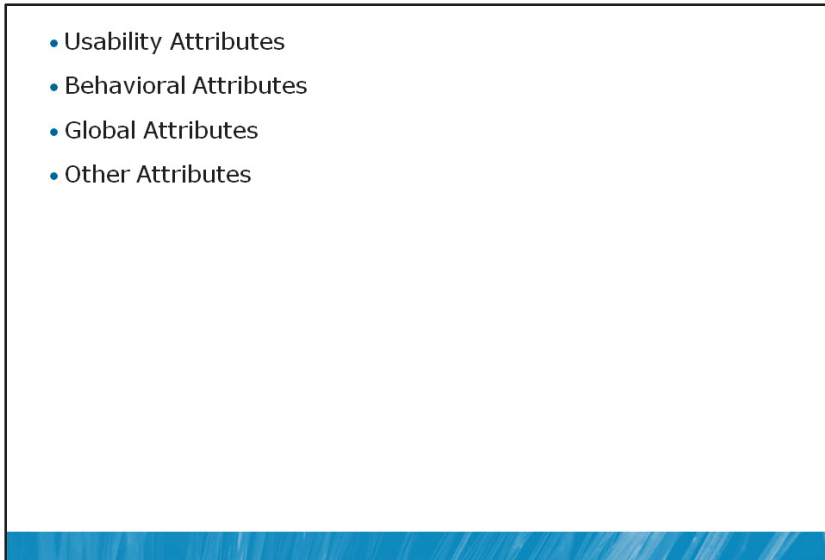
1. In the HTML file's body element after the last `</br>` element, add the following elements:

```
<label for="txtAge">Number: </label><input type="number" name="txtAge" min="0"
max="120" /><br/>
<label for="txtAge">Range: </label><input type="range" name="txtAge" min="0"
max="120" /><br/>
<label for="txtCarColor">Color: </label><input type="color" name="txtCarColor"
/><br/>
```

2. Save the file as an HTML file.
3. Run the saved file.

Lesson 2

Using Form Attributes



Web forms are the primary mechanism for gathering user input in web applications. In addition to the various field types that can constitute a form, forms also provide additional services. By using forms, you can make your web application easier and more intuitive to work with, provide tips and advice to users, and perform validation. HTML5 adds a number of attributes that you can use on a form or in form fields to specify various behaviors. This lesson introduces some of these attributes and briefly discusses attributes that have been deprecated in favor of alternatives.

Lesson Objectives

- Describe the new HTML5 form attributes.
- Explain the new usability attributes.
- Explain the new behavioral attributes.
- Explain the new global attributes.
- Use deprecated attributes.

Usability Attributes

- The **placeholder** attribute displays instructions inside text fields
- The **autofocus** attribute sets the keyboard focus when the page loads
- The **autocomplete** attribute instructs the browser to remember field values
- The **list** attribute suggests a list of auto-completion values

```
<input type="text" placeholder="Enter your full name" name="FullName" />
<input type="text" name="FullName" autofocus="autofocus" />
<input type="text" name="Username" autocomplete="on" />
```

You can use the HTML5 attributes to make forms that are easier to use.

The placeholder Attribute

Users often encounter forms they are not sure how to fill; the instructions may be unclear, the field names ambiguous, or the layout confusing. A common and effective technique to solve this problem is by adding text fields that display brief instructions or advice inside the field. HTML5 adds the **placeholder** attribute to implement this technique.

The **placeholder** attribute can be placed on any textual field (including number fields) to instruct the user on how to fill the field. Most browsers display the **placeholder** value faded out in the field when it is empty, to help users understand the purpose of the field. As soon as the field receives focus or the user starts typing text, the placeholder text disappears.

Using the **placeholder** attribute has the following advantages:

- The placeholder text is never set as the actual value of the field, so the web developer does not have to use JavaScript to remove the value before the form is submitted, or filter it out on the server afterwards.
- The web developer does not have to use JavaScript to display the instructions when the field is empty, or remove it when the user types a field value.
- The placeholder text is displayed for any textual field type, even if the field does not support freeform text. For example, the number field displays the placeholder text, even though it does not permit non-numeric entry. Another example is the password field, which also displays the placeholder text, even though it ordinarily masks the text value.
- Because the placeholder text is not the field value, validation never actually sees the instructions, so it can safely ignore it, simplifying any special JavaScript validation code.

To use the placeholder, add the attribute to a form field and write the instruction.

Using the placeholder Attribute

```
<input type="text" placeholder="Enter your full name" name="FullName" />
<input type="password" placeholder="Type your password" name="Password" />
<input type="number" placeholder="Enter your age" name="Age" />
```

You should consider the following guidelines when working with the placeholder:

- The placeholder text should be very short, so it can fit in the field. Placeholder text that is too long will probably be cut off. If longer instructions are required, the **title** attribute can be used. Most browsers render the **title** attribute as a tooltip. Longer text that needs to be visible should be placed near the field.
- The placeholder text must not contain any line breaks. The text must be single-line text with no new lines or carriage returns.
- The placeholder should not be used instead of the field label. Not every field can use the placeholder (for example, radio buttons), and not every browser supports it. The **label** element should be used for this purpose.

Prior to the introduction of the **placeholder** attribute in HTML5, web developers used the **value** attribute of the text field to display instructions. As soon as the user started typing text in the field, the instructions would disappear. Usually this technique used JavaScript to change the text style while it displayed instructions to make it clear that the text was not the actual field value. This technique is error-prone and has a few disadvantages that the **placeholder** attribute solves:

- If the user does not type anything into the field, the instructional text in the field is submitted as the field value to the server. Web developers can either use JavaScript to remove the value before submission, or filter it out on the server after submission.
- Adding validation support may make the JavaScript or server-side code more complex because the validation code would have to determine that a certain value (the instructions) represents an "empty" value.
- Using this technique with text fields that only support non-freeform text values might not be possible. For example, HTML5 does not allow non-numeric text in number fields (`<input type="number">`).

The autofocus Attribute

The **autofocus** attribute specified on a form field instructs the browser to place the focus in that field as soon as the document is loaded. When there is a field with the **autofocus** attribute, the browser places the keyboard focus in that field, enabling the user to start typing with the keyboard as soon as the page loads. This attribute makes the page easier for users who prefer the keyboard over the mouse, and more accessible for users who have difficulty in using the mouse.

The **autofocus** attribute is a Boolean attribute, which means that it acts as a toggle: on or off.

Because only one field can have keyboard focus, web developers should only set the attribute on one field. The HTML5 specifications do not mention what will happen in webpages if there is more than one **autofocus** attribute (browsers might raise exceptions, might focus on the last field with the **autofocus** attribute, etc.).

The following example sets the focus on a particular field when the document loads.

Using the autofocus Attribute

```
<input type="text" name="FullName" autofocus="autofocus" />
```


In some cases, browsers supporting the **autofocus** attribute may ignore the attribute. This can happen for several reasons, including, but not limited to, the following reasons:

- The form with the autofocus field is in a **frame** or **iframe**.
- The user may start filling out a different field before the page finishes loading. Taking the focus away from the field the user is busy filling out may be disconcerting to the user, so in this case, browsers may ignore the attribute.
- The field may be disabled or unable to receive focus for other reasons.

The **autofocus** attribute is designed primarily for pages whose main content is a form. This could be a user logon or registration form, a survey or a poll, or a corporate data-entry form. Using the attribute for forms such as these makes it clear to the user where they should start filling in the form. Setting the **autofocus** attribute on the first field in the form is a recommended good practice and helps to avoid confusing users.



Best Practice: Web developers should use the **autofocus** attribute on a case-by-case basis and should not automatically use it on every form. Consider the following guidelines to determine when and where the **autofocus** attribute is appropriate:

Pages may have more than one form. Because the attribute can be specified on only one field on a page, the web developer should choose carefully between multiple forms. This can be especially difficult when a webpage is composed of multiple disparate parts (for example, views and partials in ASP.NET MVC, or controls in ASP.NET web forms) on the server. It may make more sense in some cases not to use the attribute at all.

- When a field has keyboard focus, the user cannot use the keyboard to scroll through the page. For example, if **autofocus** is set on a search field on a news site, this can be detrimental to the user experience when reading articles that are longer than one screen.
- The **autofocus** attribute may be appropriate for search fields on sites that use search as a primary navigation mechanism, such as search engines or portals.

The autocomplete Attribute

Many web forms have similar fields and users are often asked to fill in the same information on many forms. For example, users may be asked to fill in their first and last name on many site registration forms. Requiring users to fill the same information repeatedly can be tiring and detrimental to the experience. In addition, users often make mistakes, even on common values like their names.

When the **autocomplete** attribute is placed on an **input** element, it instructs the browser whether it can remember values the user enters, and suggest them the next time the user fills a similar field. After the user fills in his or her first and last name, the next time the browser encounters a form that requests the user's first or last name, the browser can display the previous values, enabling the user to select them with the mouse or keyboard, instead of typing them into the field. Auto-completion is a usability feature because it saves the user time and helps prevent common mistakes.



Note: There are no specific constraints on the values browsers can suggest. Most often, browsers suggest values from fields with the same **name** attribute value, so after a user types his or her first name, it will be suggested the next time that user fills in a field with the same field name. However, browsers can use other criteria, for example, the browser may suggest values

based on the field type also. So a URL, an email address, or a number that was used to fill in fields with very different names may be suggested when a similarly typed field is encountered.

The **autocomplete** attribute can be used with two different types of elements: **form** elements and form field elements (for example, **input** elements). When the **autocomplete** attribute is placed on a form field element, it can have one of the following values:

- **on**—The browser may store the value the user fills in the field, and it may offer suggestions from previous forms.

The following example turns on auto-completion.

Enabling Auto-Completion for a Form Field

```
<input type="text" name="Username" autocomplete="on" />
```

- **off**—The browser is instructed not to remember the value, or offer suggestions.

The following example turns off auto-completion.

Disabling Auto-Completion for a Sensitive Form Field

```
<input type="text" name="CreditCardNumber" autocomplete="off" />
```

- **default**—When the attribute is not set, the browser uses the value of the **autocomplete** attribute of the form the field belongs to. If the **form** element does not specify an **autocomplete** value, the browser can default to "on". In other words, if the **autocomplete** attribute is not specified anywhere, the default behavior of the browser is to enable auto-completion. In addition, if it is specified on both the **form** element and a form field element, the form field will always override the **form** element, so if auto-completion is disabled for a whole form, a child field that enables auto-completion still overrides the form setting.

The following example demonstrates various auto-completion inheritance scenarios.

Examples of Form Fields Inheriting Auto-Completion

```
<form><!-- the fields will support auto-completion -->
<input type="text" name="FirstName" />
<input type="text" name="LastName" />
<input type="text" name="Age" />
</form>
<form autocomplete="yes"><!-- the fields will support auto-completion -->
<input type="text" name="FirstName" />
<input type="text" name="LastName" />
<input type="text" name="Age" autocomplete="no" /><!-- will override the form's setting -
-->
</form>
<form autocomplete="no"><!-- the fields will not support auto-completion -->
<input type="text" name="FirstName" />
<input type="text" name="LastName" />
<input type="text" name="Age" autocomplete="yes" /><!-- will override the form's setting
-->
</form>
```



Best Practice: The **autocomplete** attribute has some security implications because the browser may remember old values.

Browser storage is not guaranteed to be secure. So sensitive bank account or credit card number fields, for example, should usually have **autocomplete** set to "no".

- Browsers may be used as public terminals in Internet cafes, libraries, and other public places where many people may encounter the same forms. It is usually the responsibility of the café or library to disable the auto-completion feature on all the browsers under its purview, but this cannot be guaranteed. This is another reason to explicitly disable auto-completion with the **autocomplete** attribute, especially on sensitive fields.



Note: Browsers can ignore or override the **autocomplete** setting, enabling or disabling the setting. This usually takes the form of a global browser option the user can configure. Web developers should not rely on the auto-completion feature availability. If auto-completion is a basic requirement, the web developer should develop a suitable implementation by using JavaScript.

The list Attribute and the datalist Element

The **list** attribute provides another mechanism for auto-completion. Whereas the **autocomplete** attribute instructs the browser to suggest values based on historical entries that the browser itself manages, the **list** attribute gives the web developer more control over the suggestions. A list of pre-defined values can be specified in a **datalist** element somewhere on the page, and referenced by any text field. The browser then uses the pre-defined values as suggestions when the user fills the text field. To specify the suggestions made available by a **datalist** element, use **option** elements.

The value of the **list** attribute is the ID of the **datalist** element. A **datalist** element contains **option** elements representing its values.

The following example shows a simple text field where the list attribute and a datalist element are used to offer auto-completion suggestions to the user.

The list attribute and its corresponding datalist element

```
<input type="text" name="country" list="countries" />
<datalist id="countries">
  <option value="Canada" />
  <option value="England" />
  <option value="France" />
  <option value="Germany" />
  <option value="Japan" />
  <option value="Russia" />
  <option value="United States" />
</datalist>
```

The previous example used the **value** attribute of the **option** element. It is also common to specify options by using simple text content.

The following example also supports auto-completion.

Data List option Elements with Text Content

```
<input type="text" name="country" list="countries" />
<datalist id="countries">
  <option>United States</option>
  <option>Canada</option>
  <option>England</option>
  <option>France</option>
```

```
<option>Germany</option>
<option>Russia</option>
<option>Japan</option>
</datalist>
```

One of the advantages of using the **list** attribute with a **datalist** element is that you can create a combo box, in which the user can select an item from a list, or type a value directly in the box. This overcomes a known limitation in previous HTML implementations, where web developers could either add a text field for freeform text entry, or a **select** element that renders as a drop-down list with a finite set of values, but they could not easily combine them. To create a combo box, web developers would use text fields and complex JavaScript code to emulate this behavior.



Note: Browsers are not required to show only values specified in the associated **datalist** element. Browsers are permitted to add their own suggestions. To disable the browser suggestions, set the **autocomplete** attribute to "off". Turning off auto-completion for a field disables browser suggestions, but does not disable auto-completion with **datalist** values. However, disabling auto-completion while using the **datalist** is generally not recommended because it usually limits the user's options unnecessarily.

Legacy Support

Not every browser supports the **datalist** element. For this reason, the **datalist** element is designed to interact safely with the **select** element. The list of values is specified as a set of **option** elements for both the **datalist** element and the **select** element. When the **datalist** element contains a **select** element, which in turn contains the **option** elements, they can share the same list.

When newer browsers encounter the **datalist** element, they extract the nested **option** elements, from any depth, and ignore everything else, including the **select** element. Browsers that recognize the **datalist** element do not visually render its contents, so the **select** element will never be rendered.

However, older browsers that do not recognize the **datalist** tag skip over the tag and try to render the element contents. This behavior is not unique to the **datalist** element; it is the way browsers are designed to perform when encountering any unrecognized tags.

By placing a **select** element inside a **datalist** element, and placing the **option** elements inside the **select** element, older browsers can show a conventional drop-down list. Newer browsers can show a combo box.

Graceful Fallback for the datalist Element

```
<label>
  Enter your country:
  <input type="text" name="country" list="countries" />
</label>
<datalist id="countries">
  <label>
    Or select a country from the list:
    <select name="country">
      <option value=""></option>
      <option>United States</option>
      <option>Canada</option>
      <option>England</option>
      <option>France</option>
      <option>Germany</option>
      <option>Russia</option>
      <option>Japan</option>
    </select>
```

```
</label>  
</datalist>
```

The example demonstrates a few important things:

- The **input** element and the **select** element have the same **name** attribute value. This is a convenient way to send whatever value the user enters in either the text field or the drop-down list in older browsers. However, it is not necessary; the fields can each have different names. In any case, the web developer would probably include JavaScript validation code, ensuring that the user does not fill in both fields.
- Browsers that recognize the **datalist** element do not render its contents, so the second **label** element, the **select** element, and the **option** element are hidden. Browsers that do not recognize it show not only the **select** drop-down list, but also the **label** instructions.

Behavioral Attributes

- The **multiple** attribute toggles multi-valued input fields
- The **form** attribute separates the form behavior from its layout
- The **formaction** attribute can override the form **action** attribute

```
<input type="file" name="photos" multiple="multiple" />
<input type="text" name="FullName" form="theForm" />
<input type="submit" value="Create Template" form="editForm"
formaction="/CreateTemplate" />
```

The multiple Attribute

Form fields generally let users enter a single value for any given field. For example, a website that requires users to provide personal information generally expects each user to have a single first name, a single last name, and a single date of birth. As web applications become more complex, they tend to require support for multiple values for several field types. For example, a photo album site may allow users to upload a number of image files at the same time or a web-based email client may enable users to specify multiple recipients at the same time. To support such behavior, web developers usually use complex techniques. For example, the photo album site could have used JavaScript to add file fields dynamically, and the email client could have used complex JavaScript code to make a single field behave as though it has multiple values.

HTML5 introduces the **multiple** attribute to support and simplify this behavior. When the **multiple** attribute is specified on an **input** element, the browser recognizes that the field supports multiple values, and it adjusts the UI accordingly. The **multiple** attribute can only be used with file and email fields (**input** elements whose **type** attribute is either **email** or **file**).



Note: The **multiple** attribute is a Boolean attribute: when it is absent, it is equal to **false** and the feature is turned off; when it is present, it is equal to **true** and the feature is turned on.

For example, the photo album site can replace all the complex JavaScript code with a single declared input element whose type is "file" and has the multiple attribute.

A file field with the multiple attribute

```
<input type="file" name="photos" multiple="multiple" />
```

Supporting browsers generally show a different file selection dialog box that supports selecting multiple files.

Instead of using complex JavaScript to support multiple recipients, for example, the web developer only specifies the **multiple** attribute.

An email field with the multiple attribute

```
<input type="email" name="recipients" multiple="multiple" />
```

When the **multiple** attribute is specified, supporting browsers can provide additional services that would usually be difficult to implement. For example, if auto-completion is enabled for the “recipients” field, the browser can support auto-completion for each value entered, instead of just the first entered value.

The form Attribute

When a web form is submitted, the browser collects all the field data and sends it to the resource identified in the **action** attribute of the form. Prior to HTML5, the only way to associate a field with a form was to contain it within the **form** element. One advantage of this constraint is that it permitted multiple forms to be defined on a page. Each form contained its own fields, so there was no danger of overlap.

However, the constraint also has some disadvantages. For example, the layout requirements for a site could require different fields to be located far from each other on the page, or they might require the buttons to appear on a menu that cannot easily be placed in a shared container with the rest of the form. To ensure all the field values get submitted together, either the form layout would have to be changed or complex JavaScript or CSS would have to be developed. A common solution for this problem, employed for example by ASP.NET web forms, was to wrap all of the page content with a single **form** element.

Another, more complicated example is a form that could be used to add, modify, or delete some data. Because a single form can only declare one **action**, submitting the form would unnecessarily send all the data on the form to the server, even when not all of the data is required, as is usually the case when deleting a record. Deletion usually requires just an identifier. An easy solution is to create a second form on the page. The second form has a different **action**, a **hidden** identifier field and a **submit** button. When it is submitted, only the identifier is sent to the server. The problem with this approach is that it makes the page layout very difficult because related buttons are often placed close to each other in the UI. Because forms should not be nested in each other, solving this common problem would also require complex CSS or JavaScript.

HTML5 introduces the **form** attribute, which can be placed on any form field element. The **form** attribute value is the ID of its **form** element. When the **form** attribute is placed on an element, that element no longer has to be wrapped inside its **form**.

A field with the form attribute can be anywhere on the page.

A free-floating input element

```
<input type="text" name="FullName" form="theForm" />
<form id="theForm" method="post" action="/foo">
<input type="submit" value="Send it in!" />
</form>
```

This can make the layout easier. Submit buttons can be placed on toolbars outside their forms, and fields can be located in different areas on the page.

Solving the layout problem for multiple forms, as in the preceding deletion example, is also simpler.

Simple layout for multiple forms with a common toolbar

```
<menu id="toolbar">
<input type="submit" value="Save" form="editForm" />
```

```

<input type="submit" value="Delete" form="deleteForm" />
</menu>
...
<form id="editForm" method="post" action="/Edit">
<input type="hidden" value="SomeIdentifier" name="id" />
<label>Title: <input type="text" name="title" value="The Title" /></label>
<label>Content: <textarea name="content">The Content</textarea></label>
<label>Files: <input type="file" name="files" multiple="multiple" /></label>
</form>
<form id="deleteForm" method="post" action="/Delete">
<input type="hidden" value="SomeIdentifier" />
</form>

```

In the preceding example, the “editForm” form could contain a lot of data. Submitting it to delete it could waste a lot of bandwidth and could take more time than necessary. So the “deleteForm” form only contains the identifier, the bare minimum information needed to perform that action. Both buttons are located on a toolbar elsewhere on the page and are associated with their respective forms by using the **form** attribute.

The formation Attribute

There are cases where a single form is designed to be used for more than one action. For example, the “editForm” could be used to edit an existing blog post or to save the blog post as a template to make future blog posts easier to write. Both “actions” require the same data, but perform different operations. Prior to HTML5, one action would ordinarily be used, but that action would check what operation was being requested, which is inelegant and prone to error.

The new HTML5 **formation** attribute solves this problem. When the **formation** attribute is placed on a **submit** button, clicking the button executes the action defined in the button **formation** attribute instead of the form **action** attribute. This enables creating multiple buttons that perform different operations on the same form. The **formation** attribute can only be used with **submit** and **image** buttons.

Another toolbar button can be added that uses the “editForm” form to perform different operations.

Using the formation Attribute

```

<menu id="toolbar">
<input type="submit" value="Save" form="editForm" />
<input type="submit" value="Create Template" form="editForm" formation="/CreateTemplate"
/>
<input type="submit" value="Create a Copy" form="editForm" formation="/Clone" />
<input type="submit" value="Delete" form="deleteForm" />
</menu>

```

In the preceding example, the **formation** attribute is used with the **form** attribute, providing additional flexibility. A button can be placed anywhere on the page and perform any operation, and still be associated with the form.

Just as the **formation** attribute can be used to override the form **action** attribute, the **formmethod** and **formnovalidate** attributes can be used in exactly the same way to override the **method** and **novalidate** attributes of their associated **form** elements. Similar to the **formation** attribute, the **formmethod** and **formnovalidate** attributes can also be specified on submit buttons. The **novalidate** attribute is covered in the next lesson.

Global Attributes

- The **draggable** and **dropzone** attributes simplify drag-and-drop
- The **hidden** attribute hides unused elements until they are required
- The custom **data-*** attributes are useful for storing runtime data

```
<div draggable="true">This div can be dragged!</div>
<div dropzone="move s:text/plain">This div supports move operations for
plain text</div>
<section id="login" hidden="hidden"></section>
<article id="task2" data-position="1"></article>
```

HTML5 introduces a number of global attributes that are especially useful with forms. Global attributes can be used with any element, not just **form** elements. This topic briefly introduces some of the HTML5 global attributes.

Drag-and-Drop Attributes

HTML5 adds better support for drag and drop with the Drag-and-Drop API, which is covered in a later module. For now, it is important to note that prior to HTML5, drag-and-drop operations would ordinarily be implemented by responding to mouse events and manually handling each stage of the drag-and-drop operation, a tedious and error-prone process. HTML5 makes the drag-and-drop implementation not only easier, but also mostly implementation agnostic. In other words, the Drag-and-Drop API is designed to support drag-and-drop on various kinds of devices, including devices that do not use the mouse, such as smartphones.

To facilitate drag-and-drop, HTML5 adds two attributes to be used with the Drag-and-Drop API: the **draggable** and **dropzone** attributes. The **draggable** attribute is designed to work with the **dragstart** event and the **dropzone** attribute is designed to work with the **drop** event. The events are not demonstrated here, but their relationship to the attributes is explained.

The draggable Attribute

The **draggable** attribute instructs the browser whether an element can be dragged. The **draggable** attribute can be set to "true" or "false", or it can be absent. If the **draggable** attribute is "true", the user may drag the element; if "false", the user may not drag the element. When the attribute is unspecified, the browser resorts to the default behavior for the type of element. The **a** element (with an **href** specified) and the **img** element are draggable by default. Text, even when not wrapped by an element, in some situations may also be draggable by default.

The following example specifies that a div can be dragged by the user.

Using the draggable Attribute

```
<div draggable="true">This div can be dragged!</div>
```

When a user starts dragging a draggable element, the **dragstart** event is raised. The JavaScript code in the **dragstart** event handler is responsible for storing certain information about the drag operation on the **dataTransfer** object. The object stores the following information:

- A string describing the data type of the content being dragged (usually a mime type string)
- The kind of drag operation that is being executed (move, copy, or link, explained shortly)
- Data

The dropzone Attribute

The **dropzone** attribute indicates that content can be dropped on an element. The **dropzone** attribute also describes, to a very limited extent, the kind of content that can be dropped on the element. In many cases, JavaScript may also be required to determine if content can be dropped on the element.

When content is dropped on the element, the **drop** event is raised. The **drop** event handler is where the web developer implements the drop operation. The purpose of the **dropzone** attribute is to *quickly* determine whether the element can *potentially* serve as a drop target for the dragged content.

The **dropzone** attribute value is composed of two parts:

- The kind of drag operation the element supports (move, copy, or link)
- A list of data types that can be dropped on the element

The operation can be a "copy", "move", or "link" operation. These are broad categories describing all drag-and-drop operations. When content is dragged, it is either being copied somewhere, moved from one place to another, or being referenced (that is, linked to). Desktop browsers often display a different icon for each kind of operation, as the content is dragged over drop zones. This provides the user with valuable visual feedback. Selecting the kind of operation is usually left to the web developer.

The second part is a data types list that describes the kind of content that can be dropped on the drop zone element. The list may be empty, which indicates that there are no specific *declared* constraints. JavaScript code may be used to filter the content, whether constraints are specified or not. When content is dragged over an element that specifies supported data types, one of the data types must match the data type specified on the **dataTransfer** object initialized in the **dragstart** event handler. If there are no matches, the content cannot be dropped on the element.

Supported data types are specified as a list separated by spaces where each data type is preceded by an "s" or an "f" and a colon (":"). The "s" specifies a string data type and the "f" specifies a file data type. These are usually MIME type strings, although any string without spaces can be used. For example, to specify that a PNG image file can be dropped on an element, the constraint would be specified as "**f:image/png**". To support multiple image file types, they can be separated by spaces: "**f:image/png f:image/jpeg f:image/bmp**". To specify the data as a simple string, the data type may be written as "**s:text/plain**".

The following example demonstrates different dropzone attribute usages.

Usages of the dropzone Attribute

```
<div dropzone="copy f:image/png">This div supports copy operations only for PNG image files</div>
<div dropzone="copy f:image/bmp f:image/gif">This div supports copy operations for both BMP and GIF image files</div>
<div dropzone="move s:text/plain">This div supports move operations for plain text</div>
```

The hidden Attribute

The **hidden** attribute is a Boolean attribute used to instruct the browser not to render an element. Although elements can also be hidden by setting the **display** CSS property to "none", the two methods for hiding elements have very different purposes.

- To improve the user interface to help users, the **display** property is used to control the visual presentation of the webpage. This can be used to selectively display or hide paragraphs, form fields, tabs, and other elements. For example, not every country uses a state or region designation, so the **display** property can be used to display the "state" field when a relevant country is selected from the list, or hide it otherwise. Another common example is a webpage that contains tabs, where only the information of the selected tab is displayed. The **display** CSS property is appropriate for these scenarios, because it helps users fill forms easily, or present information in a manner that is more user-friendly.
- The **hidden** attribute also affects the visual presentation, but its purpose is to help web developers control the application flow. For example, new web applications are increasingly implemented as "one-page sites", where the user interacts with a single webpage (at a single URL) that displays different screens by using JavaScript, without actually navigating to a new webpage (or URL). In such a web application, the **hidden** attribute can be applied to all screens, except the active screen, and JavaScript can be used to navigate between screens by toggling the **hidden** attribute when the active screen changes.

The following example shows the hidden attribute used to specify the inactive screens.

Using the hidden Attribute

```
<body>
<section id="home">
<header>
<h1>Home Screen</h1>
</header>
<p>This is the home screen.</p>
<button onclick="showLoginScreen()">Log In</button>
<button onclick="showProductsScreen()">Products</button>
</section>
<section id="login" hidden="hidden">
<header>
<h1>Login Screen</h1>
</header>
<form action="/login.aspx">
<p><label>Username: <input type="text" name="username" /></label></p>
<p><label>Password: <input type="password" name="password" /></label></p>
<input type="submit" value="Log in" />
</form>
</section>
<section id="products" hidden="hidden">
<header>
<h1>Products Screen</h1>
</header>
  This screen displays the company's products.
</section>
</body>
```

The preceding example contains three **section** elements, each representing a screen. The second and third **section** elements have the **hidden** attribute, so they are not rendered when the page loads. The first **section** represents the "home page" and has two buttons, one to activate the login screen and the other to activate the products screen. Only one screen is rendered at a time, but the URL never changes.

The **hidden** attribute is also useful for accessibility. Screen readers can use the **hidden** attribute to determine which elements are active. For example, when the logon screen in the preceding example is displayed by removing the **hidden** attribute, screen readers can detect the change and inform their users. In other words, the **hidden** attribute does not hide an element only from view, the attribute prevents the element from being rendered in every type of browser, including browsers that help people who are blind or have low vision.

The data-* Attributes

When a browser encounters an attribute that it does not recognize, it ignores the attribute. However, the attribute remains accessible through the DOM. Web developers often store runtime information on the elements. Sometimes, the extra attributes are generated on the server; at other times, they are generated by client-side JavaScript code.

As the technique became more popular, naming conflicts arose between third-party tools and websites. For example, some browser add-ons modify the HTML on the fly, adding non-standard attributes that might conflict with similarly-named attributes used by websites. In addition, some websites and third-party tools used attributes that were eventually added to the HTML standard.

To avoid naming conflicts and to standardize the usage of this technique, HTML5 introduces the **data-*** attributes. Web developers can affix their own attributes to the DOM elements, as long as their attribute names are prefixed with **data-**. For example, if a web developer wants to add an "index" attribute to DOM elements, the full name of the attribute becomes "**data-index**".

Third-party tools and browser extensions are forbidden from using **data-*** attributes and are responsible to avoid naming conflicts with the HTML standard.

The **data-*** attributes can be specified on any element, but they are very commonly used with form elements. They can be used, for example, to store validation information that JavaScript code can use to validate a form before it is submitted. Another example is to store the original element order in a list that users can manually sort, and which can be used to undo the user's changes.



Best Practice: Web developers are encouraged to use the **data-*** attribute to store interactive data. Web developers should avoid storing custom attributes that do not use the **data-** prefix.

The following example demonstrates the use of custom **data-*** attributes.

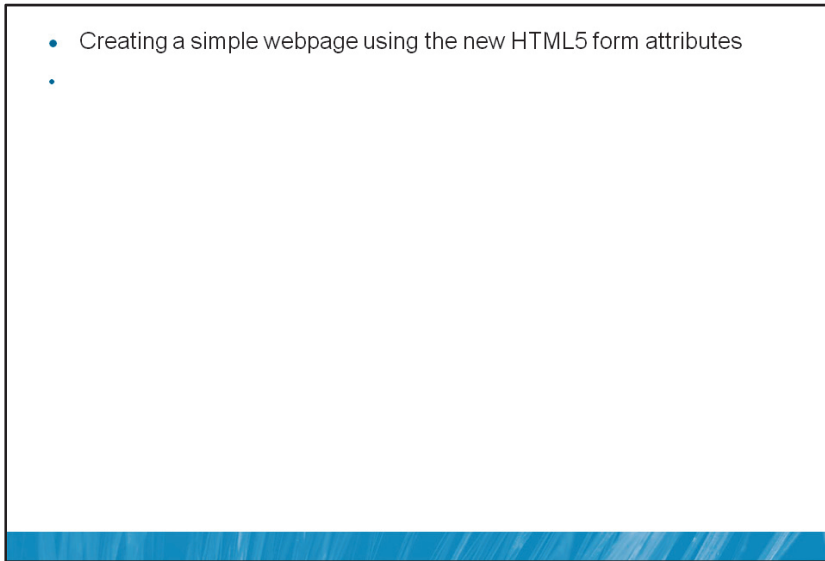
Using the data-* Attributes

```
<section id="tasks">
  <article id="task1" data-position="2">
    <header>
      <h1>Task #1</h1>
    </header>
    These are the details for task #1.
  </article>
  <article id="task2" data-position="1">
    <header>
      <h1>Task #2</h1>
    </header>
    These are the details for task #2.
  </article>
  <article id="task3" data-position="3">
    <header>
      <h1>Task #3</h1>
```

```
</header>  
  These are the details for task #3.  
</article>  
</section>
```

The preceding example contains three **article** elements, each displaying information for a task in a hypothetical task list. The **id** attribute of each **article** element is the identifier of the associated task. The **data-position** attribute is a custom attribute used by the web application to store information about the on-screen position of the task. If the tasks are re-ordered (possibly by using drag-and-drop or special buttons to move tasks up or down), JavaScript code can be used to update the new position. Using the **data-position** attribute is preferable to using a **position** attribute, which could introduce naming conflicts with third-party tools.

Demonstration: Using Form Attributes



This demonstration shows how to:

- Create a new Empty Web Application in VS2010.
- Add a new HTML file.
- Add markup to the web page using the new HTML5 form attributes.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file's body element add the following elements:

```
<body>
  <form>
    <input type="text" placeholder="Enter your full name" name="FullName"
autofocus="autofocus" autocomplete="on" />
    <input type="password" placeholder="Type your password" name="Password" />
    <input type="number" placeholder="Enter your age" name="Age" />
    <input type="text" name="country" list="countries" />
    <datalist id="countries">
      <option value="Canada" />
      <option value="England" />
      <option value="France" />
      <option value="Germany" />
      <option value="Japan" />
      <option value="Russia" />
      <option value="United States" />
    </datalist>
    <input type="file" name="photos" multiple="multiple" />
  </form>
  <section id="products" hidden="hidden">
    <header>
      <h1>Products Screen</h1>
    </header>
    This screen displays the company's products.
```

```
</section>  
</body>
```

4. Save the file as an HTML file.
5. Run the saved file.

Other Attributes

- The **contentEditable** Attribute
- Deprecated Attributes

```
<div contenteditable="true"></div>
```

The contentEditable Attribute

The **contentEditable** attribute determines if an element can be edited. When **contentEditable** is set to "true" or to an empty string (for example, `contenteditable=""`), the browser places the element in a special state in which the element contents can be edited as though the element is a word editor. The mechanism is extensible and commands can be added and controlled by using JavaScript. Any content supported by HTML can be theoretically typed into an editable element, including text with different fonts, styles, and colors.

Although the attribute can be used on any element and not just **form** elements, it enables the editable element to accept user input. Using JavaScript, the content of the editable element can be persisted and recalled at a later date.

The following example shows a div whose content can be edited.

Usage of the contentEditable Attribute

```
<div contenteditable="true"></div>
```

In the preceding example, the cursor can be placed inside the div, and the user can start typing to edit the div content.

When the **contentEditable** attribute is set to "false", the element content cannot be edited, even if the element is inside another element that can be edited.

Deprecated Attributes

HTML5 not only added new elements and attributes; it also removed some attributes. The following list describes some of the HTML4 attributes that have been deprecated in HTML5. These are not specifically form attributes.

- The **align** attribute is deprecated in favor of the CSS **text-align** property.

- The **background** attribute of the **body** element is deprecated in favor of the CSS **background-image** property.
- The **text** attribute of the **body** element is deprecated in favor of the CSS **color** property.
- The **link**, **vlink**, and **alink** attributes of the **body** element are deprecated in favor of the CSS **color** property on the various pseudo-classes of the **a** element.
- The **bgcolor** attribute is deprecated in favor of the CSS **background-color** property.
- The **border** attribute is deprecated in favor of the CSS **border-width** property.
- The **clear** attribute is deprecated in favor of the CSS **clear** property.
- The **language** attribute of the **script** element is deprecated in favor of the **type** attribute.
- The **nowrap** attribute is deprecated in favor of the various CSS text-wrapping properties.
- The **start** attribute of the **ol** element is deprecated.
- The **type** attribute of the **ol**, **ul**, and **li** elements is deprecated in favor of the CSS **list-style-type** property.
- The **value** attribute of the **li** element is deprecated.
- The **version** attribute of the **html** element is deprecated in favor of the DTD.
- The **width** and **height** attributes **td** and **th** elements are deprecated.
- The **width** attribute of the **pre** element is deprecated.

Lesson 3

Validation

- Form-Level Attributes
- Range Validation Attributes
- General Validation Attributes

Good validation can be difficult to implement in the browser. Web applications can place many validation requirements on a web form. Every data type has its own restrictions (for example, numbers have different restrictions than strings); every business has its own rules (for example, “the user must be 13 or above”); databases, operating systems, and browsers have their own limitations (for example, some databases restrict Unicode strings to 4,000 characters); and security considerations add more constraints to the data.

Performing validation in the browser, on the client-side, has several advantages:

- It helps users enter correct data, that is, data complying with a set of rules.
- It helps users understand what kind of data is required. Forms can be ambiguous and confusing to users, so validation (along with other usability features) can help clarify the form.
- It reduces the number of roundtrips to the server required for validation, because some validation is performed before the form is submitted.
- It can show immediate interactive feedback to the user with instructions on how to correct or improve the data.



Best Practice: Validation in the browser is very useful, but is not a replacement for validation on the server. Form validation rules can be manipulated in the browser by users, and forms that are submitted to servers can be intercepted and modified during submission. As a general rule, every validation that is done on the client should also be done on the server. Very often, the server performs additional validations.

Many concerns are alleviated by simply showing fields that restrict the input to structurally valid values, such as a number field that only allows legal numbers, or a Boolean field represented by a check box. HTML has always offered a number of **input** elements to support various data types. As already shown, HTML5 adds support for more field types, which means that more data that can be described by one of those types is less likely to be susceptible to mistaken data entry.

However, adding field types does not help much with ambiguity, business rules, and interactive feedback. JavaScript was originally created to facilitate using code to perform validation with the intent to solve these very problems. JavaScript code can prevent or initiate form submittal, implement business rule validation, and provide feedback to the user.

Even JavaScript is not the ultimate solution. Some browsers do not support JavaScript, and some users and businesses disable it on browsers that do support JavaScript. So, JavaScript cannot be relied on to provide validation.

HTML5 adds new attributes that are used with form fields to help resolve some validation issues. None of these attributes require JavaScript, because they are implemented natively by the browser. This lesson covers some of the new validation-specific attributes.

Lesson Objectives

After completing this lesson you will be able to implement validation by using:

- Form-level validation attributes
- Range validation attributes
- General validation attributes

Form-Level Attributes

- Form-level attributes (**novalidate**, **formnovalidate**)
- The **novalidate** and **formnovalidate** attributes can selectively disable form validation before submitting the form

```
<form id="loginForm" novalidate="novalidate">
  <input type="text" name="username" required="required" />
  <input type="password" name="password" required="required" />
  <input type="submit" value="Log In" />
</form>
```

The novalidate Attribute

Usually, the browser prevents an invalid form from being sent to the server. However, in some cases, the form must be sent even with invalid data:

- If the user tries to delete a record, it does not matter that the record contains invalid data.
- When storing work-in-progress, it may be prudent to store the unfinished work, even if some of its data is invalid, so it can be resumed at a later time.
- The web developer may want to implement custom validation to replace browser validation.

The **novalidate** attribute enables forms submission, even when without validation. Otherwise, the browser may prevent submittal if the form contains invalid data. The **novalidate** attribute is placed on the **form** element and is a Boolean attribute. Even when the attribute is present, the browser may continue to validate data entered into the form and provide immediate feedback as the data is entered.

The following example demonstrates the use of the **novalidate** and **formnovalidate** attributes.

Usage of the novalidate and formnovalidate Attributes

```
<form id="loginForm" novalidate="novalidate">
  <input type="text" name="username" required="required" />
  <input type="password" name="password" required="required" />
  <input type="submit" value="Log In" />
</form>
<menu id="toolbar">
  <input type="submit" form="editForm" value="Save Draft!" formaction="/save/draft"
  formnovalidate="formnovalidate" />
  <input type="submit" form="editForm" value="Save" />
</menu>
<form id="editForm" action="/save/permanent">
  <input type="text" name="title" required="required" />
  <textarea name="content"></textarea>
</form>
```

The preceding example contains two forms:

- The first form, with the **"loginForm"** ID, has the **novalidate** attribute. When the user clicks the submit button, the form will be submitted, even if the **username** and **password** fields are empty, despite these fields being mandatory fields.
- The second form, with the **"editForm"** ID, does not have the **novalidate** attribute, so if the mandatory **"title"** field is empty, it is not submitted. The **menu** element has two buttons. The first button saves a draft version of the content. To support saving a draft version, even when the mandatory **"title"** field is empty, the button has the **formnovalidate** attribute. The second button submits the form if it is valid.

Range Validation Attributes

Range validation attributes:

- The **min** and **max** attributes specify range boundaries
- The **step** attribute defines the increment size inside the range

```
<input type="range" min="0" max="100" step="5" /> <!-- 0,
5, 10, 15, ..., 95, 100 -->
```

The min and max Attributes

The **min** and **max** attributes are used to specify a valid range of values for an **input** element. When one or both attributes are specified, the field value cannot be less than the **min** attribute value or more than the **max** attribute value. Both attributes are optional and can be used independently of each other.

The min and max attributes can be used with various field types, as shown in the following example.

Usage of the min and max Attributes

```
<input type="number" min="-10" max="10" />
<input type="number" min="-10" />
<input type="number" max="10" />
<input type="time" min="9:00" max="17:00" />
<input type="range" min="0" max="100" />
<input type="date" min="2000-01-01" max="2019-12-31" />
```

In addition to validating that the value is in the range specified by **min** and **max**, the browser can also prevent users from entering out-of-range values. For example, when a browser renders a number field with **min** equal to 0, pressing the down arrow widget that decreases the field value does not reduce the field value below 0.

The step Attribute

The **step** attribute defines the increment sizes between valid values. For example, if the **step** for a number field is 10, the valid values are divisible by 10. The number 7 would be invalid. If the **min** attribute is specified, the smallest value is the value of the **min** attribute and the **step** attribute starts counting from the **min** attribute. So if the **step** is 10 and **min** is 5, the valid values will be in the series: 5, 15, 25, 35, and so on.

The following examples show the use of the step attribute.

Usage of the step Attribute

```
<input type="range" min="0" max="100" step="5" /><!-- 0, 5, 10, 15, ..., 95, 100 -->
```

```
<input type="number" min="5" step="10" /><!-- 5, 10, 15, 25, ... -->
```

When the **step** attribute is used for **time** input fields, it specifies seconds and is written as a regular whole number, instead of the time format.

If a field control in the browser can constrain entry, it may take the **step** attribute into consideration. For example, if a number field has **step** equal to 10, pressing the up or down widgets increases or decreases the field value by 10, respectively.

The step attribute is written a bit differently with time input fields, as the following example shows.

Usage of the step Attribute with a time input field

```
<input type="time" min="9:00" max="17:00" step="300" /><!-- 300 seconds = 5 minutes -->
```

General Validation Attributes

Validation attributes:

- The **required** attribute is used to specify mandatory fields
- The **pattern** attribute can specify regular expressions
- The **maxlength** attribute limits the number of characters

```
<input type="text" name="username" required="required" />
<label> Zip Code: <input type="text" pattern="\d{5}" name="zip" /></label>
<input type="text" name="FirstName" maxlength="10" />
```

The required Attribute

The **required** attribute designates that a field must be filled before its form can be submitted. The attribute is a Boolean attribute that can be specified for **input**, **select**, and **textarea** elements. When it is specified on a field whose value is empty, the form cannot be submitted.

The required attribute indicates that the field must be filled before the form can be submitted, as the following example shows.

Using the required Attribute

```
<input type="text" name="username" required="required" />
```

The pattern Attribute

The **pattern** attribute specifies a *regular expression* that describes a valid value. A regular expression is a string that uses special symbols to describe a pattern that can be matched against other strings.

 **Note:** Regular expressions are beyond the scope of this course, but the following example gives an idea of what they can do. There are many references and tutorials available in print and on the Internet.

Using regular expressions helps ensure that users not only enter the correct data type, but also that the data is structured correctly. For example, there is no field type for zip codes. If a form requires a zip code, the **pattern** attribute can be set to the pattern `"\d{5}"`, which would indicate that a five-digit zip code is expected, where each digit must be in the range, 0 to 9. Because regular expressions provide such a fine level of control, almost any string pattern can be specified.

Any regular expression can be used with the pattern attribute, as the following example shows.

Usage of the pattern Attribute

```
<label> Zip Code: <input type="text" pattern="\d{5}" name="zip" /></label>
```



Note: Note that the regular expression only validates non-empty fields. To validate against empty fields, add the **required** attribute.

The maxlength Attribute

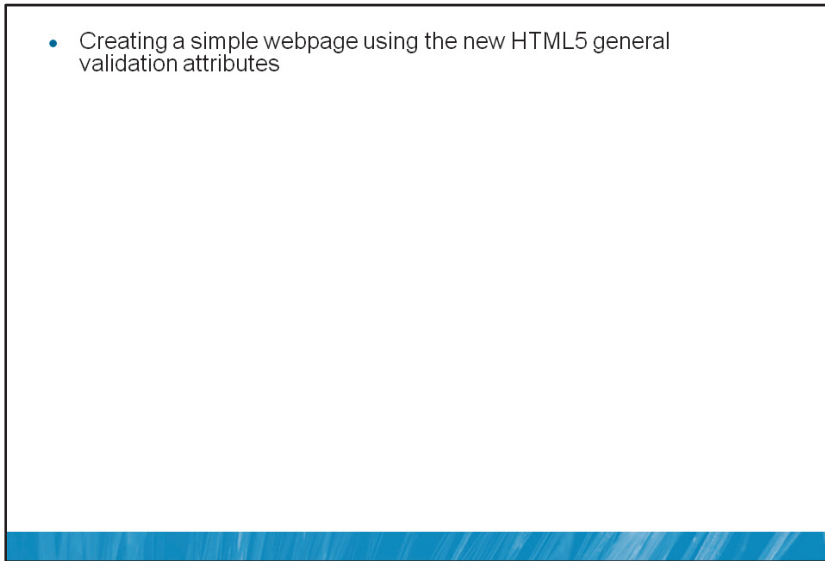
The **maxlength** attribute specifies the maximum number of characters the user can type in a textual field. This includes standard **textinput** elements, **textarea** elements, and the new **url** and **emailinput** fields. If the number of characters in the field exceeds the value of the **maxlength** attribute, the field (and the form) is invalid.

The maxlength attribute can limit the number of characters in a textual field, as the following example shows..

Usage of the maxlength Attribute

```
<input type="text" name="FirstName" maxlength="10" />
```

Demonstration: General Validation Attributes



This demonstration shows how to:

- Create a new Empty Web Application in VS2010.
- Add a new HTML file.
- Add markup to the web page using the new HTML5 general validation attributes.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file's body element add the following elements:

```
<body>
  <form>
    <input type="text" name="username" required="required" />
    <label> Zip Code: <input type="text" pattern="\d{5}" name="zip" /></label>
    <input type="text" name="address" maxlength="10" />
    <input type="submit" name="submit" value="Submit The Form" />
  </form>
</body>
```

4. Save the file as an HTML file.
5. Run the saved file.

Lesson 4

Using Browser Detection, Feature Detection, and Modernizr

- Cross-Browser Interoperability
- Browser Detection
- Feature Detection
- Modernizr
- Using HTML5 Polyfills

HTML5 is developed very rapidly. On the other hand, the browsers are developed fast and so some HTML5 features are not supported in some browsers. This situation makes developing HTML5 websites or applications more complex.

There are some techniques web developers can use to help address the complexity caused by the unpredictable HTML5 feature support, such as browser and feature detection, or the use of polyfills. This lesson introduces these methods along with the Modernizr JavaScript library, which is one of the common libraries to detect HTML5 features.

Lesson Objectives

After completing this lesson you will be able to:

- Describe the need for cross-browser interoperability
- Use browser detection
- Use feature detection
- Apply feature detection by using the Modernizr JavaScript library
- Use polyfills for backwards compatibility

Cross-Browser Interoperability

- One of the main problems with earlier versions of HTML was cross-browser interoperability
- Developers strive for code that will work in all browsers
 - This is impossible because of legacy browsers and the difference between browser engines

In the early days of the World Wide Web, when there were only two browsers in the browser market, creating websites that simply worked was easily achievable. The reason is obvious—there were only two ways to render a webpage. Over the years, however, the browser market grew and more browsers were created. With the lack of strict specifications in earlier versions of HTML (versions 1-4), each browser vendor created its rendering engine and added new functionality that was not a part of the specifications. This situation complicated the work of web developers, who now needed to adapt their websites and applications to every major browser available.

Because of the complexity involved in creating websites and applications with markup to account for every possible browser, many companies decided to make their websites compatible with a single browser only. This solution alienated a large number of potential users. Today, this problem of interoperability is a known issue, addressed with some techniques such as browser and feature detection. These techniques help web developers determine which browser the user is currently running and which features should be utilized to achieve functionality.

Browser Detection

- A technique to check which browser is running the code
- Can make dozens of assumptions based on this single check
- Should be avoided most of the time
- Should only be used when applying code to a specific legacy browser of a known version

```
<!--[if IE 7]><div>rendered in Internet Explorer 7 only</div><![endif]-->
```

Browser detection is used to determine the user agent of the browser that the user is currently using to view the webpage. This technique helps the web developer choose which functionalities to activate for the user. This technique was frequently used in the past, when there were fewer browsers, but is being abandoned these days. This is because browser detection makes dozens of assumptions about the browser capabilities based on the result of a single check; if even one of these assumptions is wrong, it can break the webpage. This is why browser detection should be generally avoided. The only situation in which browser detection is appropriate is when code is written for a specific legacy browser of a known version.

Browser detection can be achieved by using conditional comments.

- The standard for creating an HTML comment is `<!-- a comment -->`. Internet Explorer uses conditional comments, which are written as comments with an inner block of HTML.

The following conditional comment checks if the browser is Internet Explorer 7; if it is Internet Explorer 7, the conditional comments renders the internal div element.

Conditional Comment

```
<!--[if IE 7]><div>rendered in Internet Explorer 7 only</div><![endif]-->
```

Only if the browser fulfills the conditions will the inner block of HTML be rendered, along with its scripts and CSS, if any. Other browsers will not render the inner HTML because it is commented out.



Additional Reading: For more information about conditional comments, go to the following link: [Conditional Comments](#)

- Detecting unique objects:
Many browsers have their own unique objects that are not part of the standards. Detection of these objects can imply that a specific browser is currently viewing the webpage. For example, the `document.all` object is unique to Internet Explorer. The developer will put an if statement on that

object to determine its existence. For example, `if(document.all) { }`. Only if that if statement returns "true", the developer will create a behavior with `document.all`. Using unique objects can be risky, however, because other browsers might add the object also.

- **User agents:**
The navigator JavaScript objects have a `userAgent` property that holds the user-agent header sent by the browser to the server. Using this property can help to detect which browser is being used. For example, running the `navigator.userAgent` code inside the IE Developer Tools console in Internet Explorer 9 displays the following output: "Mozilla/5.0 (compatible; MSIE 9.0; Windows NT 6.1; Trident/5.0; SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center PC 6.0; OfficeLiveConnector.1.3; OfficeLivePatch.0.0; .NET4.0C; .NET4.0E; InfoPath.3; Tablet PC 2.0; MS-RTC LM 8; Zune 4.7)"



Best Practice: Browser detection should only be used when code is written for a specific legacy browser. Otherwise, feature detection should be used.

Feature Detection

- A technique to check the availability of a feature before it is used
- Standard features should be looked for first
 - Browsers sometimes support both the standards and a legacy implementation
 - Looking for standard features first will ensure the use of standards if available
- Avoid making any assumptions about existing features

```
if (window.addEventListener) {
  // HTML that will be rendered if addEventListener is available
}
else {
  // HTML that will be rendered if addEventListener is not available
}
```

Feature detection is very common today. This technique checks whether a desirable feature (an object, function, property, or some type of behavior) is available in the browser before using it. If the feature does not exist in the browser, you can use other features as a backup or simply omit the functionality that you wanted to implement.

When using feature detection, you should check standards first. This is because the new browsers sometimes support the standard implementation alongside a legacy implementation; pointing to the standards first ensures that the legacy code is not used if the standard has not been implemented. For example, the standards include the use of the **window.addEventListener** function, which is supported in Internet Explorer 9. Internet Explorer 9 also supports the **window.attachEvent** function for backward compatibility with older versions of Explorer.

When using feature detection, do not make any assumptions. If one feature is detected, it in no way implies that other features, even those connected to the detected feature, are also available in the browser. For example, the existence of the **postMessage** function does not imply that the **addEventListener** function is also available.

The following example shows how to use feature detection to detect if **addEventListener** is available.

Feature Detection Example

```
if( window.addEventListener )
{
  // HTML that will be rendered if addEventListener is available
}
else
{
  // HTML that will be rendered if addEventListener is not available
}
```

HTML and CSS both use feature detection in some fashion, by default. Some of the new HTML5 elements, for example, are simply ignored by browsers that do not support them. This means that you can place fallback content inside elements that might not be supported.

The following example shows how to use a video element with fallback content.

HTML Fallback Content

```
<video src="test.mp4">
  <object src="test.mp4">
    <a href="test.mp4">
      Download The Test Video
    </a>
  </object>
</video>
```

In the preceding example, if the browser supports the HTML5 video element, then the object inside it will be ignored. If the browser does not support the HTML5 video element, then the object inside it will be used by the browser and the video will still be rendered and displayed.

In CSS, unrecognized properties are ignored. You can therefore use this behavior to detect features. Remember, you should first check for standards.

The following example shows how to use feature detection in CSS.

Feature Detection in CSS

```
.roundedbox
{
  border-radius: 10px;
  -ms-border-radius: 10px;
  -moz-border-radius: 10px;
}
```

The preceding example illustrates how to use feature detection in CSS. You will first put the property that is stated in the specifications. If the browser does not support the property, the browser ignores the property and goes to the next property. The `-ms` and `-moz` are prefixes that are used by browsers to add their own implementation until a feature is made a part of the specifications. `-ms` is a prefix used in Internet Explorer and `-moz` is a prefix used in FireFox.

Most JavaScript libraries today, such as jQuery, use feature detection. This grants the libraries cross-browser capabilities and helps them to apply feature detection to most of the browsers.

Modernizr

- A small JavaScript library
- Detects availability of HTML5 and CSS3 specifications and whether features are natively implemented in the browser
- Uses feature detection under the hood

```
if (Modernizr.canvas) {
    // HTML that will be rendered if the Canvas element is available
}
```

Modernizr is a small JavaScript library that you can download from <http://go.microsoft.com/fwlink/?LinkID=241689>. It is also available in the web project templates such as ASP.NET MVC 3 of Visual Studio 2010. Modernizr helps detect the availability of HTML5 and CSS3 features natively implemented in the browser.

How to get started with Modernizr?

First, download the library from the Modernizr website: <http://go.microsoft.com/fwlink/?LinkID=241690>. During the process of downloading the library, you will choose which features you want to detect from the list of available ones, or you can choose to detect all the features. After you download the library, you can integrate Modernizr in two different ways to the project. You can:

- Add it to your website or application.
- Create a new web application project in Visual Studio 2010, which adds the Modernizr library to the Scripts folder, by default.

After Modernizr is added to the project, you will need to reference it in the webpages that will use it.

You can reference Modernizr by adding the following script tag into the head element of the webpage.

Adding the Modernizr Script Reference

```
<head>
...
<script src="modernizr.js" type="text/javascript"></script>
</head>
```



Note: Note that in the preceding example, the Modernizr file name is modernizr.js. When you download Modernizr, it will have a version number in the file name.

How does Modernizr detect features?

When a webpage incorporating Modernizr is loaded, Modernizr first runs tests to determine if each of the selected features exists. It will create elements in memory, set specific instructions, retrieve those instructions, and check the returned values, determining if the requested features exist in the browser or not, based on the result.

The following example shows the code found in Modernizr.

Modernizr Canvas Test

```
tests["canvas"] = function() {  
  var elem = document.createElement("canvas");  
  return !!(elem.getContext&&elem.getContext("2d"));  
};
```

How is Modernizr used?

After you load the webpage and run Modernizr tests, you use the Modernizr JavaScript object to check if a desired feature is available.

The following example checks whether canvas is supported in the browser.

Modernizr Canvas Check

```
if (Modernizr.canvas)  
{  
  // HTML that will be rendered if the Canvas element is available  
}
```

Demonstration: Modernizr

- Creating a simple webpage which use Modernizr to detect HTML5 features

This demonstration shows how to:

- Create a new Empty Web Application in VS2010.
- Add a new HTML file.
- Add markup to the web page.
- Add Modernizr support to detect HTML5 features.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. Add the **modernizr-1.7.js** file from **D:\Mod03\Democode\Assets**.
4. In the html file add the following script tag inside the head element:

```
<head>
  <title>Hello Modernizr</title>
  <script src="modernizr-1.7.js"></script>
</head>
```

5. Add an empty script tag at the end of the body tag:

```
<body>
  <script type="text/javascript"></script>
</body>
```

6. In the empty script tag, add the following code:

```
if (Modernizr.canvas)
{
  alert('canvas is available');
}
if (Modernizr.video)
```

```
{  
  alert('video is available');  
}  
if (Modernizr.audio)  
{  
  alert('audio is available');  
}
```

7. Save the file as an HTML file.
8. Run the saved file.

Using HTML5 Polyfills

- Polyfills are piece of code (or plug-in)
- They provide the feature that the browser should provide *natively, if it is missing*
- They help to fill the gap when expected features are not available in a specific browser

Polyfillor shim is the common name for a piece of code that provides the features that a browser should provide natively, but the features are unavailable. Whenever native support for a feature does not exist (mainly in older browsers), if you want to provide that functionality, you will have to create a polyfill or use an external polyfill. For example, if `addEventListener` does not exist in an old browser such as Internet Explorer 8, a polyfill script can imitate that functionality for the browser and make it support such a function.

Because many features that HTML5 and CSS3 include are not supported in older browsers, polyfills are used mainly in HTML5 and CSS3.

One feature of Modernizr is its ability to fill the gaps in older browsers and add some sort of support for HTML5 elements. When you add a Modernizr reference to the webpage and it is rendered in a browser that does not support the new HTML5 elements (such as header, footer, and so on) Modernizr will enable that support. What Modernizr does underneath is to introduce those elements to the DOM and add to them the `display:block` CSS instruction to make them visible. It also adds support to some of the new CSS3 instructions, such as `box-shadow` and `border-radius`.

 **Note:** Polyfills are not a solution for every problem. There are times that you prefer to omit some functionality, instead of using a polyfill.

 **Best Practice:** Try to detect first if a feature exists by using Modernizr; if it does not exist, load a polyfill to fill the gap. The latest version of Modernizr has a `Modernizr.load` function that will help you to do exactly that.

Lab: Creating Form Input and Validation

- Creating the Checkout and Game Review Forms
- Add validation to the Checkout and Game Review forms.
- Adding Modernizr Support

Logon Information

Virtual Machine	Virtual machine: SEA-DEV
User Name	User name: Admin
Password	Password: Pa\$\$w0rd

Estimated Time: 45 minutes

Scenario

You have created most of the structure and application pages for the new Xbox Games online store. Now, you need to build the form that will enable users to buy games and add a game review. The form should make use of input elements and validation.

Objectives

After completing this lab you will:

- Develop the Checkout and Game Review data entry forms.
- Add validations to the webpages.
- Add Modernizr support to the webpages.

Exercise 1: Creating the Checkout and Game Review Forms

Scenario

In this exercise, you will build two new webpages: the Checkout page and the Game Review page. Both webpages will be created as forms that expect an input from the user.

► Task 1: Create the Checkout webpage.

1. Open the starter solution, which is located in the **D:\Mod03\Labfiles\Starter** folder.
2. Create the **Checkout.html** file.
3. Create the checkout markup by using the provided design image, **Checkout.png**, which is located in **D:\Mod03\Labfiles\Starter\Design**.

► Task 2: Create the Game Review webpage.

1. Create the **GameReview.html** file.
2. Create the game review markup by using the provided design image, **GameReview.png**, which is located in **D:\LabFiles\Starter\M03\Design**.

► Task 3: Add CSS styles to the Checkout and Game Review webpages.

1. Create the **forms.css** file.

► Task 4: Add navigation to the new webpages.

1. Add navigation to the Checkout page in the navigation bar in the header of the webpages.
2. Add navigation to the **Game Review** page from the **Game Details** page.

Results: Checkout and Game Review Forms are created by utilizing HTML5 input types.

Exercise 2: Add validation to the Checkout and Game Review forms.

Scenario

In this exercise, you will add the required validation attributes to some of the fields in the Checkout and Game Review forms. You will also add a placeholder and autofocus to some of the fields.

► Task 1: Add validation and HTML5 attributes to the Checkout webpage.

1. In the **Checkout.html** file, add the **required**, **placeholder**, and **autofocus** attributes in the buyer information fields and in the credit card information fields.

Results: The Checkout and Game Review forms with validation attributes are created.

Exercise 3: Adding Modernizr Support

Scenario

In this exercise, you will create two polyfills. You will use Modernizr to detect whether the **number** input type exists, and if not, you will create a polyfill to validate that the input for the **number** text boxes is a number. You will also use Modernizr to check if the **placeholder** attribute is supported, and if not, you will create a polyfill to copy its behavior.

► Task 1: Add the Modernizr JavaScript library to the project.

1. Add the **Modernizr** file to the **Scripts** folder. The file is located in **D:\Mod03\Labfiles\Starter** folder.
2. Add a script reference to **Modernizr** in the **head** element of the **Homepage.html**, the **Checkout.html** and **GameReview.html** webpages.

► Task 2: Create the number input type and placeholder attribute polyfills.

1. Create polyfills for number validation in the **number** input type and for the functionality of the **placeholder** attribute.

Results: The form pages to utilize the Modernizr library are created.

Module Review and Takeaways

In this module, you reviewed the creation of web forms by using the new input types in HTML5 and using the new attributes incorporated into HTML5. You also learned about the form validation concept. In addition, you learned about browser detection, feature detection, and the Modernizr JavaScript library, which helps detect HTML5 features.

Review Questions

How can you use HTML5 today, when there are browsers that do not support some of its features?

Tools

Tool	Use for	Where to find it
Modernizr JavaScript library	Browser detection of HTML5 features.	http://go.microsoft.com/fwlink/?LinkID=241689

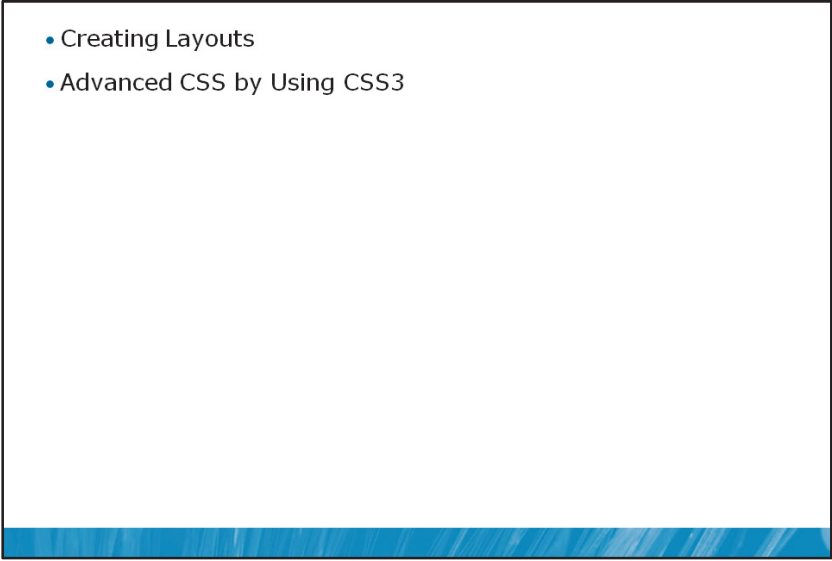
Module 4

Laying Out and Styling Webpages

Contents:

Module Overview	4-2
Lesson 1: Creating Layouts	4-3
Lesson 2: Advanced CSS by Using CSS3	4-17
Lab: Laying Out and Styling Webpages	4-35
Module Review and Takeaways	4-37

Module Overview

- 
- Creating Layouts
 - Advanced CSS by Using CSS3

A good layout can direct users to important parts of a website or application so they can easily find the content they are looking for. A poor layout, on the other hand, will drive away users. There are many aspects to the creation of good layouts that will be introduced in this module. Therefore, creating an effective layout and styling is a very important part in the process of creating a website or application. This is, in fact, one of the earliest steps that you should take, because it has important implications for the rest of the process.

In this module, you will learn important concepts about the layout and styling of webpages and applications. You will also learn how to use the new features of Cascading Style Sheets 3 (CSS3), such as new selectors, text effects, and multi-column layouts, to improve the look and feel of your webpages.

Objectives

After completing this module, you will be able to:

- Create a webpage layout.
- Apply advanced CSS to webpages.

Lesson 1

Creating Layouts

- Basic Webpage Layouts
- Placement and Position
- Using Tables for Layout
- Using div Elements for Layout
- Using CSS3 for Layout

Websites and applications are judged by their content and layout. An effective layout can make a site or application shine. A poor layout can drive users away. Good layouts, therefore, direct users to important sections and help them find what they are looking for.

The creation of layouts can be considered an art. This phase must take place very early in the process of creating a website or application because it influences many aspects of the process. This phase requires the involvement of web designers, User Experience (UX) experts, and web developers, who together decide the concepts that will form the basis for the site or application's appearance. After collating these concepts, the web designers create the design for the website or application. The design contains details about the colors, fonts, page, or section dimensions, and more. After the design is ready, the web developer implements the layout.

A layout can be implemented in many ways. In the early days of site design, table layouts ruled the web, because they were easy to create, but they imposed a lot of problems such as performance problems and page size. Later, developers began using the floating **div** layouts, which are still used today. Today, the preferred method for laying out a webpage is to use HTML5 structural elements combined with **div** elements. CSS3 introduces many new CSS properties that help to create better layout with the new HTML5 elements by applying columns and templates.

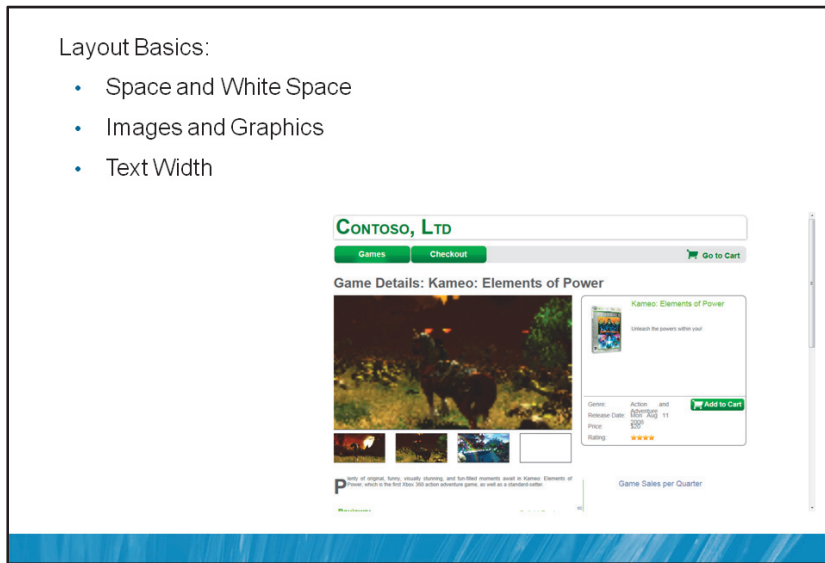
This lesson introduces the concept of layouts and how to create and use them on webpages.

Lesson Objectives

After completing this lesson, you will be able to:

- Explain the concepts behind basic webpage layouts.
- Explain the concepts of placement and position in webpage layouts.
- Create webpage layouts by using tables.
- Create webpage layouts by using the **div** element.
- Create webpage layouts by using the new CSS3 layout options.

Basic Webpage Layouts



The basic concepts of layout might seem trivial, yet can be very powerful when applied correctly. Proper use of space and white space, images and graphics, and text width can be the difference between a good-looking website or application, and an ugly website or application.

Space and White Space

When you look at a webpage, the elements that direct you to important sections are ones you probably do not even notice: space and white space. Using white space can make a website or application look very elegant. Space and white space include two concepts: macro white space and micro white space.

- Macro white space is a large space located between major blocks of content. Breaking the webpage into such blocks provides room for each individual block and helps users identify the different sections.
- Micro white space is the space between letters and list elements, within a block of content. Using line-height or letter-spacing can improve the readability of the content; therefore, the value of micro white space should not be underestimated.

Images and Graphics

Images and graphics, such as vector graphics, clip art, and bar graphs, can catch the user's eye. Overusing them, however, can distract the user from the main content of the webpage. Think of a webpage that was filled with graphical elements. Did the large number of images and graphics help you read the content or only distract you? A good webpage is one that strikes a balance between graphics and content. Images can improve the layout of a page, but they must be used with proportion and care.

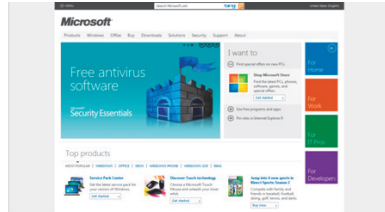
Text Width and the Number of Words in a Line

Text width, font size, and word length, determine how many words will be displayed on one line of text. Putting too many words on a line can reduce readability. Putting too few words can distract the reader. It is important to find a balance in the numbers of words on a line. You can use styles such as font size, line height and margins to affect this balance, but they can be hard to apply if the text area is limited.

Although most of the time you, as a developer, won't necessarily be the one to make the decisions about the layout and design, you still need to know why it is so important. When you are in doubt about the use of a layout or a design, consult the user-experience experts and designers for guidance.

Placement and Position

- Placement and position must not be made randomly:
 - Some elements are related to the webpage goal
 - Some elements are related to other elements on the webpage
- Good placement helps direct users to points of interest on the website or application



When you start to create a webpage, it is empty. You begin to fill that empty page with content. Positioning elements on the webpage is a very important process because it determines which elements will grab the user's attention.

The first task is to determine the elements that should be present on the webpage. You will probably need a header, footer, navigation bars, and so on. These are the easiest elements to create and position, because they are commonly used on just about every website and application. Some other questions you might ask yourself may include:

- Where will you place a paragraph, a section, or a div that contains important content?
- Would you place an image that is not related to the page's content, such as an advertisement, in the middle of the webpage?
- Where will you place images that are related to your content? Will the images be placed near that content or elsewhere?

The preceding questions are a few of the questions that can help you understand how to place various elements on a webpage.

Purpose of the Element

Understanding the purpose of every element is a prerequisite for placing each element. For example, you should not place a header element at the lowermost part of the page. Every web developer knows where to place webpage elements like headers and footers, because their placement is related to their role. Other examples are headlines, which should not be placed at the lowermost part of a page, and navigation elements, which should be placed at the top or to the left or right. Have you ever seen a navigation bar in the middle of a webpage?

Relationship to Webpage Goals

Understanding the goal of the webpage can help you determine where to place various elements. For example, a checkout page will include form elements that help collect the relevant information from the

user. There will also be a summary of the cart on that page, but because it is only related to the goal of the webpage, it will be located near the form elements.

Relationship between Elements in the Web Page

When there is a relationship between elements on the webpage, you should place them close to one another. For example, an image placed inside a paragraph implies a direct relationship between the image and the paragraph content. Conversely, horizontal lines separate two elements on a webpage and imply that they are not connected.

Points of Interest within a Webpage

The placement of points of interest on a webpage can direct users either towards them or away from them. Putting white space around an important section, for instance, affects how it is seen and emphasizes its importance.

Ask yourself the following questions:

- What is the purpose of each element?
- What is the relationship between the elements on the page?
- What are the main points of interest on the webpage?

Not making use of white space, positioning, and changes in text size can cause the user to become disoriented. This reduces the usability of the Web page.

Consider this example webpage.

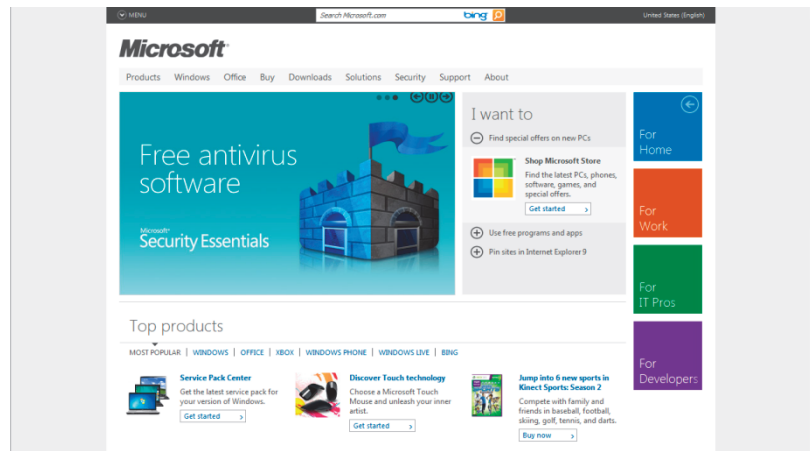


FIGURE 4.1: GOOD WEBPAGE EXAMPLE

The use of white space leads the user to immediately notice the product, makes the location of the navigation bar clear, and sets off the related content at the lowermost part.

Using Tables for Layout

Tables for layout:

- The simplest way to lay out a Web page
- Should be used only for the presentation of tabular data
- Tables are not a layout tool

HTML tables were created for tabular data display, but were abused by designers in the early days of web development to create grid layouts for webpages. By simply setting the table border attribute to 0, you had a grid surface that rendered a table for layout.

The following code shows a simple webpage that uses an HTML table for its layout.

Table as Layout

```
<!doctype html>
<html>
<head>
...
</head>
<body>
<table border="0">
<tr>
<td colspan="2">
<h1>Page Title</h1>
</td>
</tr>
<tr>
<td style="width:200px; ">
Menu
</td>
<td style="width:400px;">
Page Content
</td>
</tr>
<tr>
<td colspan="2">
Footer
</td>
</tr>
</table>
</body>
</html>
```

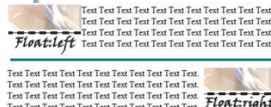
The preceding example demonstrates how the page layout is divided into a header (first **tr**), body (second **tr**), and footer (third **tr**).

You should avoid the use of tables for layout as much as possible. The reason it is being presented in this course is that in real-world web development, you may find yourself working on a webpage that already uses a table for its layout. If this happens, you should refactor it and use another means of providing the layout.

Using div Elements for Layout

- **Div** elements are containers used for grouping HTML elements
- Combining **divs** with CSS make it possible to lay out and position **the** elements using:
 - Floating
 - Positioning
- **Div** elements are commonly used today
- Should be used along with HTML5 structural elements

```
<!doctype html>
<html>
<head></head>
<body>
  <div>
    <div><h1>Page Title</h1></div>
    <div>
      <div style=" width:200px;float:left;"> Menu</div>
      <div style="width:400px;float:left;">Page Content</div>
    </div>
    <div style="clear:both;">Footer</div>
  </div>
</body>
</html>
```



One alternative to using tables for layout is to use **div** elements. **Div** elements are block-level elements that represent a division (or section) of the webpage. They can be nested inside other **div** elements and can be floated and positioned by using CSS. This means that by combining **div** elements with CSS, you can create a more efficient and effective layout.

The following example shows div elements used to create the layout presented in the preceding topic.

Layout with Div Elements

```
<!doctype html>
<html>
<head>
  ...
</head>
<body>
  <div>
    <div><h1>Page Title</h1></div>
    <div>
      <div style=" width:200px;float:left;"> Menu</div>
      <div style="width:400px;float:left;">Page Content</div>
    </div>
    <div style="clear:both;">Footer</div>
  </div>
</body>
</html>
```

As you can see, this layout has fewer elements and is based only on **div** elements and their block behavior. **Div** elements are widely used for layout today. Although writing layouts with **div** elements is more complex than using tables, it provides better webpage design. When using **div** elements for layout, webpages are written with fewer elements, are downloaded faster, and are better understood by search engines.

However, with the availability of the new HTML5 structural elements, which are block-level elements similar to the **div** element, laying out pages has become easier for the web developer. You can now create your layout by using elements more meaningful than **div** and get the same results. This does not mean

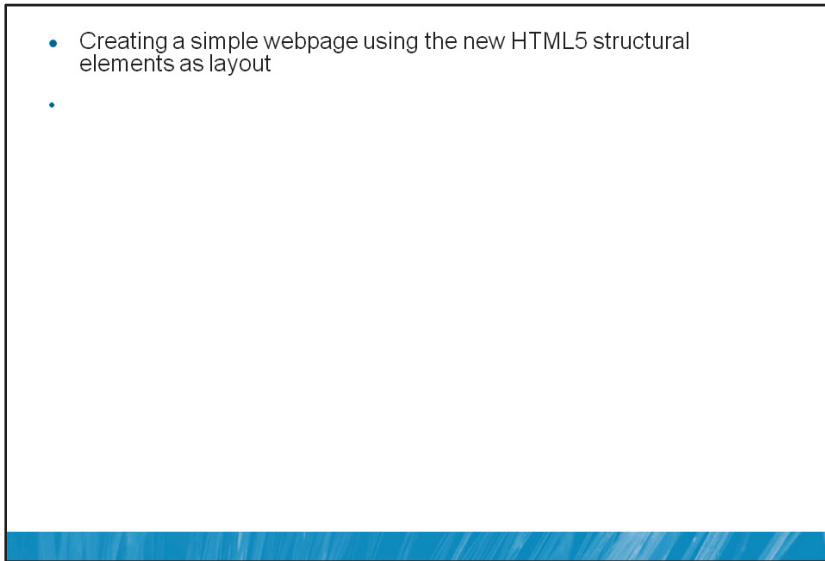
you should stop using **div** elements, but there are new options in HTML5 that are more useful in certain circumstances.

In the following code from the preceding example, HTML5 structural elements are used, instead of div elements.

Layout with the New HTML5 Elements

```
<!doctype html>
<html>
<head>
  ...
</head>
<body>
<div>
<header><h1>Page Title</h1></header>
<div>
<menu style=" width:200px;float:left;"> Menu</menu>
<section style="width:400px;float:left;">Page Content</section>
</div>
<footer style="clear:both;">Footer</footer>
</div>
</body>
</html>
```

Demonstration: Using HTML5 Structural Elements for Layout



This demonstration shows how to:

- Create a new Empty Web Application in VS2010.
- Add a new HTML file.
- Add markup to the web page using the new HTML5 structural elements as layout.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file's body element add the following elements:

```
<body>
  <div>
    <header><h1>Page Title</h1></header>
    <div>
      <menu style=" width:200px;float:left;"> Menu</menu>
      <section style="width:400px;float:left;">Page Content</section>
    </div>
    <footer style="clear:both;">Footer</footer>
  </div>
</body>
```

4. Save the file as an HTML file.
5. Run the saved file.

Using CSS3 for Layout

CSS3 introduces a number of new ways to lay out webpages

- Template-based positioning
- Multi-column

```
div
{
width: 100px;
column-width: 45px;
column-count: 3;
column-gap: 1em;
}
```

Pellentesque habitant

>Lorem ipsum dolor sit amet, consectetur adipiscing elit. Maecenas porttitor congue massa. Fusce posuere, magna sed pulvinar ultricies, purus lectus malesuada libero, sit amet commodo magna eros quis urna. Nunc viverra imperdiet enim. Fusce est. Vivamus a tellus.

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Proin pharetra nonummy pede. Mauris et orci. Aenean nec lorem.

...

The new CSS3 specifications introduce several new ways to lay out webpages. These ways make it more convenient and easy to make layouts that were previously very difficult to create. The positioning of elements on the webpage can be achieved through a number of methods, such as templates and multi-columns. In the end, the method you choose to create the layout is your responsibility, and knowing the options available will help you decide which is best to use in any situation.

Template-based positioning

Elements can be positioned by using the CSS position property, which can take the values of the following types:

- Absolute positioning. You specify a position by using horizontal and vertical coordinates.
- Relative positioning. You specify that an element is positioned relative to its parent.
- Fixed positioning. You specify that an element is positioned on a fixed set of coordinates on the webpage.
- Template-based positioning. You place an element in slots in a table-like template.

A template defines one or more slots into which an element's content can flow. The templates do not allow elements to overlap. Template-based positioning is created by using the combination of the display CSS property, which determines how the template looks, and the position property, which sets a slot in the display. By using template-based positioning, you can create very complicated layouts.

Template-based positioning can help create high-level layouts by positioning widgets (which are small web components that have some specific implementation) or creating a grid layout (layout of a grid).

The following code is an example of the use of template-based positioning.

Template-Based Positioning

```
<style>
div { display: "ab"
      "cd" }
#elm1 { position: a }
```

```

#elm2 { position: b }
#elm3 { position: c }
#elm4 { position: d }
</style>
<div>
<span id="elm1">This</span>
<span id="elm2">Is</span>
<span id="elm3">A</span>
<span id="elm4">Template</span>
</div>

```

The preceding example shows a portion of a webpage. It includes a style block that defines the template and fills the slots with content. After the style block is another portion of the webpage, which will be affected by the template style. In the example, a simple template is defined by the value of the display property. When the display of an element is set to a template, it is called template element. After creating the template, each element is positioned inside a slot. The rendering of this code might look like this:

This Is

A Template



Additional Reading: For more information about CSS template layout, see the following link: [CSS3 Template Layout](#)

Multi-column

CSS3 introduces a new way to arrange content in columns. The multi-column CSS3 module describes how to apply CSS to the content to create multiple columns, such as in a newspaper. In multi-column, content can flow from one column to another, and the number of columns will vary depending on the size of the screen. Multi-columns are a great tool for web pages or applications that will be read on devices such as smartphones and tablet PCs which have different screen sizes and resolutions.

To create the column, you use the new CSS properties of **column-width**, **column-count**, and **column-gap**.

column-width: Determines the width of the columns. This attribute can take values greater than zero, or the auto value, which means that the width will be determined by other properties.

The following is an example of using column-width.

column-width

```

div
{
  width: 100px;
  column-width: 45px;
}

```

In preceding example, **column-width** is set to 45 pixels and the **div** width is 100 px. This means that there will be two columns of 50 px each. If the width of the **div** had been 90 pixels, the two columns would be displayed with a width of 45 pixels each. If the width of the **div** were less than 90 pixels, the display of the columns would have been narrowed.

column-count: Determines the number of columns to display. This attribute takes a value greater than zero, or the auto value, which means that the number of columns will be determined by other properties.

The following is an example of using column-count.

column-count

```
div
{
  column-count: 3
}
```

In the preceding example, the **div** will be divided into three columns. For example, if the **div** is 100 pixels wide, it will hold 2 columns with a width of 33 pixels each and one column with 34 pixels width.

columns: A shorthand property that sets both **column-width** and **column-count** at once. If a value is not set for one of the two, it will automatically be set to auto.

The following is an example of using columns.

columns

```
columns: auto 20px; /* column-width: 20px; column-count: auto */
columns: 2 20px; /* column-width: 20px; column-count: 2 */
```

column-gap: Sets the gap between the columns. It can take any non-negative value; the default value allows the browser to specify its width.

The following is an example of using column-gap.

column-gap

```
body {
  column-gap: 1em;
}
```

In the preceding example, every multi-column in the body will have a gap of 1 em. The value, 1 em, is suggested for **column-gap** by the HTML5 specifications, although some browser engines may render a different default value.

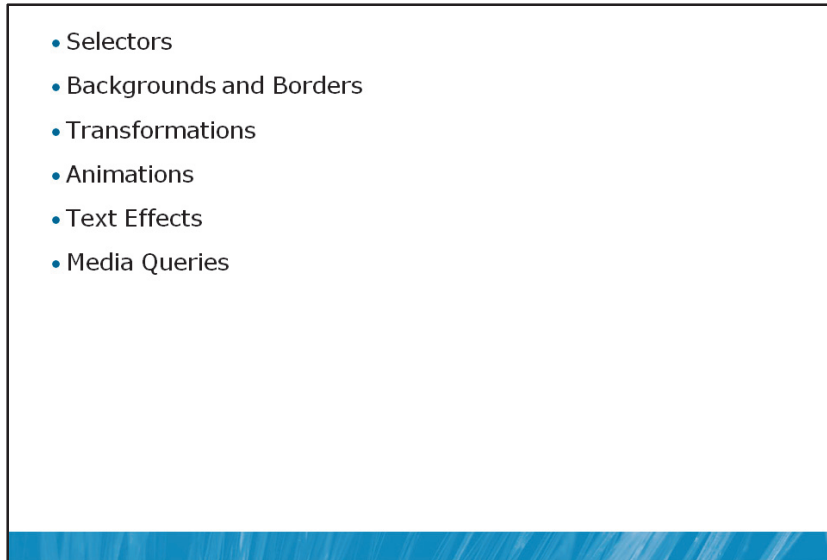
Although there are many other properties in the multi-columns module, they will not be covered in this course.



Additional Reading: For more information about multi-column module, go to the following link: [CSS3 Multi-Column](#)

Lesson 2

Advanced CSS by Using CSS3



The last CSS update was version 2.1. During the development of HTML5, the need for a new and improved CSS arose, so W3C began developing CSS3 after CSS2.1. The new version of CSS, CSS3, adds to or replaces some of the functionality of CSS2.1.

CSS3 implements a lot of changes, including new ways to lay out webpages (some of which were learned in the previous lesson), model animations, transform HTML elements, and more. These changes can help decrease the need for browser plugins and external JavaScript libraries.

In this lesson, you will learn about CSS3 and the new options it brings to the world of style.

Lesson Objectives

After completing this lesson, you will be able to:

- Apply CSS3 selectors.
- Apply CSS3 backgrounds and borders.
- Apply CSS3 transformations.
- Apply CSS3 animation.
- Apply CSS3 text effects.
- Apply CSS3 media queries.

Selectors

New CSS selectors that have been introduced include:

- Attribute selectors for **starts with**, **ends with**, or **contains**
- Structural pseudo-classes such as **nth-child**, **first-child**, and **last-child**
- The **negation** pseudo-class—**not(s)** (where s is a selector)

```
a[href^="www."]
tr:nth-child(even) { color:red; } /* will do the same as the previous
statement */
span > p:last-child { color:red; } /* a paragraph, which is the last
child of a span, will be selected and its text will be red */
```

CSS3 introduces many new CSS selectors. The new selectors open new opportunities for you to apply styles everywhere on the webpage with less writing. Some of the selectors were taken from existing libraries, such as jQuery, because the CSS working group realized that these selectors were already being adopted by many web developers. The selectors in the CSS3 specifications are arranged in logical groups (such as structural pseudo-classes and attribute selectors) to make them more readable and meaningful.

CSS3 introduced new attribute selectors, including **starts with**, **ends with**, and **contains**.

Attribute starts with selector. The attribute **starts with** selector gets a value that is checked against all the HTML element attributes to see if it matches the beginning of any of their values. The syntax is: **^=**

The following is an example of the use of the attribute starts with selector.

Attribute Starts With Selector

Attribute ends with selector. The attribute **ends with** selector gets a value that is checked against all the HTML element attributes to see if it matches the end of any of their values. The syntax is: **=\$**

The following is an example of the use of the attribute ends with selector.

Attribute Ends With Selector

```
a[href$=".aspx"]
```

Attribute contains selector. The attribute **contains** selector gets a value that is checked against all the HTML element attributes to see if it appears inside any of their values. The syntax is: ***=**

The following is an example of the use of the attribute contains selector.

Attribute Contains Selector

```
a[href*="Microsoft"]
```

With the new attribute selectors, a new group of structural pseudo-classes was introduced. This group helps you select specific elements by using the **nth-child**, **first-child**, and **last-child** pseudo-classes.

nth-child. The **nth-child()** pseudo-class is a new pseudo-class in CSS3. **nth-child()** receives as a parameter a calculation statement in the form of $an+b-1$, where a and b are numbers greater than zero. That statement is calculated for each appropriate element, selecting the child elements to which the calculation applies. The index number of the first child index is 1. The **nth-child()** pseudo-class can also receive the *odd* and *even* parameters, which impose the style on odd or even elements in the selected group.

The following is an example of the use of the **nth-child** selector.

The nth-child selector

```
tr:nth-child(2n+1) { color:red; } /* will choose every odd row in an HTML table and set
its text color to red */
tr:nth-child(odd) { color:red; } /* will do the same as the previous statement */
tr:nth-child(2n+0) { color:red; } /* will choose every even row in an HTML table and set
its text color to red */
tr:nth-child(even) { color:red; } /* will do the same as the previous statement */
```

first-child. The same as calling **nth-child(1)**. It represents the first child element of an element.

The following is an example of the use of the **first-child** selector.

The first-child selector

```
span > p:first-child { color:red; } /* a paragraph, which is the first child of a span,
will be selected and its text will be red */
```

last-child. Represents the last child element of an element.

The following is an example of the use of the **last-child** selector:

The last-child selector

```
span > p:last-child { color:red; } /* a paragraph, which is the last child of a span,
will be selected and its text will be red */
```

Another, much-anticipated pseudo-class added in CSS3 is the negation pseudo-class. This class receives a simple selector as a parameter, and applies its style to elements that are not part of the selected group.

The following is an example of the use of the negation selector.

The negation selector

```
button:not([disabled]) { background-color:red; } /* The buttons that are not disabled
will have a red background */
```



Additional Reading: For additional reading, see the following link: [CSS3 Selectors](#)

Backgrounds and Borders

- New background properties:
 - **background-size**
 - **background-origin**
 - **background-clip**
- Support for multiple background images
- Color gradient backgrounds
- New border properties
 - **border-radius**
 - **border-image**

```
div
{
  background:url(flower.png);
  background-size:100px 60px;
  background-origin: content-box;
  background-clip:content-box;
}
```

CSS3 introduces new ways for you to set element backgrounds. New properties, such as **background-size**, **background-origin**, and **background-clip**, allow you to manipulate backgrounds. You can even place multiple backgrounds on a single element.

Background-size. Sets the size of a **background-image**. This enables using a single image at different sizes on the same webpage. The property parameters, **width** and **height**, can be set in pixels or percentages.

The following is an example of the use of background-size.

The background-size property

```
/* This style applies to all divs a background with a width of 100 pixels and a height of
60 pixels */
div
{
  background:url(flower.png);
  background-size:100px 60px;
}
```

Background-origin. Using this property, you can specify how the **background-position** of a background is calculated. It takes three values: **border-box**, **padding-box**, and **content-box**. **border-box** sets the background position relative to the upper-left corner of the background border. **padding-box** sets the background position relative to the upper-left corner of the padding edge. **content-box** sets the background position to the upper-left corner of the content.

The following is an example of the use of background-origin.

The background-origin property

```
/* The position of the background is set to the upper-left corner of the content */
div
{
  background-image: url(flower.png);
}
```

```
background-origin: content-box;
}
```

Background-clip. Determines the background painting area. It can accept the same three values that **background-origin** accepts: **border-box**, **padding-box**, and **content-box**, but instead of starting to paint the background from the given position, this property crops the image to fit the box model.

The following is an example of the use of background-clip.

The background-clip property

```
/* all the divs will have a yellow background and will be clipped to the content box */
div
{
background-color:yellow;
background-clip:content-box;
}
```

Support for multiple background images: Before CSS3, an element could have only one background image. In CSS3, you can set multiple background images to the same element by using a comma to separate between the image URLs.

The following is an example of how to set two images as the background to the same element:

Multiple Background Image Support

```
div
{
background-image:url(flower1.png),url(flower2.png);
}
```

Color gradients on background: CSS3 added support for setting the background of an element to a color gradient. Gradients are images that fade from one color to another. They are widely used on websites and applications for buttons, and as background images. Using gradients can decrease the dependency on images, resulting in fewer HTTP requests to the server and better performance. There are two types of gradients: linear gradients and radial gradients, which are created by setting the **background** CSS property to the new **linear-gradient** or **radial-gradient** functions. Most browsers, however, do not yet fully support these functions.

The following is a simple example of the use of linear-gradient:

linear-gradient

```
div
{
background: linear-gradient(left, blue, green);
}
```

As the preceding example shows, the first parameter of the **linear-gradient** function is the **side-or-corner** property, which sets the starting point of the gradient line. This is followed by the **color-stop** parameters, separated by commas. The **color-stop** parameters are a list of color values, each of them followed by an optional **stop-position** in percentage or in pixels. The example does not show the use of **stop-position**.

The use of gradients and the concepts behind them will be explained in more detail in later modules.

With the new background options, CSS3 also introduces new border effects, such as **border-radius** and **border-image**.

In the past, to round the corners of elements, developers used images or JavaScript libraries. The new **border-*-radius** properties (where * is the name of one of the four corners—top-left, top-right, bottom-left, or bottom-right) and the **border-radius** shortened property implement this functionality in CSS3. In the **border-radius** property, the corner shape is defined by up to two length parameters or percentage parameters, separated by a slash; the first parameter is the horizontal radius and the second is the vertical radius. If only one parameter is provided, it applies to both radii.

The following is an example of the use of border-radius.

The border-radius property

```
border-radius: 2em 1em 4em / 0.5em 2em;

/* equivalent to */
border-top-left-radius: 2em 0.5em;
border-top-right-radius: 1em 2em;
border-bottom-right-radius: 4em 0.5em;
border-bottom-left-radius: 1em 2em;
```

There are times that developers need to use an image as the border of an element. CSS3 provides the **border-image-*** properties and the **border-image** shortened property to enable this functionality. The **border-image** shortened property receives five parameters: **border-image-source**, **border-image-slice**, **border-image-width**, **border-image-outset**, and **border-image-repeat**. Each of the **border-image-*** properties deals with an individual parameter of the shortened property.

The following is an example of the use of border-image.

The border-image property

```
border-image: url(border.png) 30 30 round;

/* equivalent to */
border-image-source: url(border.png);
border-image-slice: 30;
border-image-width: 30;
border-image-repeat: round;
```

The preceding example demonstrates how to set the property. The first parameter is the source for the image that will be used in the border. Then, you set the slice parameter, which is the inward offset of the image border. Next, you set the width of the image border, followed by indicating what type of repeat should be used: repeat, round, or stretch.

Transformations

- CSS3 transformation supports two-dimensional and three-dimensional element transformations
- Two-dimensional transformations support simple transformations such as translate, scale, and stretch
- Additional functions provide support for three-dimensional transformations

```
@keyframes myAnimation
{
  0% { right: 0px; }
  25% { right: 25px; background: blue; }
  50% { right: 50px; background: red; }
  75% { right: 75px; background: blue; }
  100% { right: 100px; }
}
div
{
  animation: myAnimation 5s linear 3s infinite alternate ;
}
```

CSS3 supports two-dimensional and three-dimensional element transformations. Transformation enables the developer to change the shape, size, and position of an element. Before transformations were introduced in CSS3, such effects required installing a plug-in such as Microsoft® Silverlight®. To perform transformations, use the transform property with transformation functions. Transformation functions can be combined to create more complicated transformations.

Two-Dimensional Transformations

The two-dimensional transformation functions include:

translate. Moves an element from one location to another. The function receives two parameters: **left** and **top**. The element moves from its current position by the amount provided in the parameters. There are also two separate functions, **translateX** and **translateY**, which each receive only one parameter and perform translations along only one axis.

The following is an example of the use of the translate function.

The translate function

```
/* The div's position is translated from its current position 25 pixels left and 25
pixels top */
div
{
  transform: translate(25px, 25px);
}
```

rotate. Rotates an element. The function receives a parameter in degrees, and rotates the element clockwise by the given amount. If a negative value is given, the element rotates counter-clockwise.

The following is an example of the use of the rotate function.

The rotate function

```
/* The div will rotate 60 degrees clockwise */
```

```
div
{
    transform: rotate(60deg);
}
```

scale. Scales the size of the element. This function receives two parameters: **width** and **height**. The element size is multiplied by the given parameters. The function can also receive a single parameter, in which case it will be used to multiply both width and height. There are also two separate functions that scale the width or the height individually, called **scaleX** and **scaleY**.

The following is an example of the use of the scale function.

The scale function

```
/* The div will be scaled up until it is twice its original width and four times its
original height*/
div
{
    transform: scale(2,4);
}
```

skew. Skews an element. The **skew** function receives two parameters: *x-axis degrees* and *y-axis degrees*. The element is skewed by the specified angles, first along the x-axis, and then along the y-axis. If only one parameter is provided, it is applied to the x-axis alone, and the skew along the y-axis is 0 (no skew). There are also two separate functions that skew along only one axis each, and receive only one parameter: *skewX* and *skewY*.

The following is an example of the use of the skew function.

The skew function

```
/* The div will skew first 60 degrees on the x axis and then 60 degrees on the y axis */
div
{
    transform: skew(60deg, 60deg);
}
```

matrix. The matrix function combines all the transformation functions into one. It receives six parameters, enabling the HTML elements to which it applies to rotate, scale, translate, and skew all at once. The six parameters are entered into a transformation matrix [a b c d e f] that is applied to the element. Transformation matrices are part of linear algebra, which studies vector/linear spaces.

The following is an example of the use of the matrix function.

The matrix function

```
/* The div will not be changed at all, because the identity matrix is being applied to it
*/
div
{
    transform: matrix(1, 1, 1, 1, 1, 1);
}
```

Three-Dimensional Transformations

Three-dimensional transformations are effected by adding a z-axis to two-dimensional transformations discussed in the preceding topic. A separate set of functions have been made available to implement this

functionality.

The three-dimensional transformation functions include:

matrix3d. Receives 16 parameters, as opposed to the six parameters of the **matrix** function, describing a 3D transformation on the element.

translate3d. Receives an additional parameter above the two received by the **translate** function, moving the element on the z-axis. An additional translate function, **translateZ**, receives only one parameter and moves the element by the provided amount on the z-axis.

scale3d. Receives an additional parameter above the two received by the **scale** function, scaling the element up or down by the given value along the z-axis. An additional scale function, **scaleZ**, receives only one parameter and scales the element on the z-axis.

rotate3d. This function receives four parameters to define a three-dimensional rotation, as opposed to the single parameter of the **rotate** function. The first three parameters define the [x,y,z] direction vector around which the element will rotate clockwise by the number of degrees provided in the fourth parameter. An additional rotate function, **rotateZ**, receives only one parameter and rotates the element clockwise by the amount around the z axis.

Demonstration: Transformations

- Creating a simple webpage using the new CSS3 transform

This demonstration shows how to:

- Create a new Empty Web Application in VS2010
- Add a new HTML file
- Add markup to the web page using the new CSS3 transform

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file's body element add the following elements:

```
<div id="element"></div>
```

4. In the HTML file's head element add the following style tag:

```
<style type="text/css">
#element
{
    position: absolute;
    left: 50px;
    top: 50px;
    height: 20px;
    width: 20px;
    background-color: Red;
    -ms-transform: rotate(30deg) scale(1.5, 2);
}
</style>
```

5. Save the file as an HTML file.
6. Run the saved file.

Animations

- CSS3 introduces a method for creating simple animations
- Effectively replaces animations usually created with animated images, Microsoft® Silverlight®, and JavaScript
- Uses the **@keyframes** rule and **animation** property

```
@keyframes myAnimation
{
  0% { right: 0px; }
  25% { right: 25px; background: blue; }
  50% { right: 50px; background: red; }
  75% { right: 75px; background: blue; }
  100% { right: 100px; }
}
div
{
  animation: myAnimation 5s linear 3s infinite
  alternate ;
}
```

Animation is one way to draw attention to components of a website or application. Today, most websites contain some sort of animation, whether in an advertisement, dashboard, graph, or another place. By using animation, you create a more interactive and attractive website. Before CSS3 animations, however, even simple animations needed to be implemented by using animated images, or through plug-ins such as Silverlight or JavaScript.

The CSS3 animations module describes how to create simple animations. The animation is applied to an element or elements, effecting a gradual style change over a time frame. This decreases the amount of JavaScript used for animation and lets the browser handle the animation. Animations in CSS3 also remove the need to install a browser plug-in just to implement a simple animation, such as moving an element from one position to another.

To create a CSS3 animation, first create an **@keyframes** rule that defines the start point, frames, and the end point of the animation. The **@keyframes** rule definition should be followed by an identifier, which will be used as the animation name.

The following is the definition of an @keyframes called myAnimation:

@keyframes Definition

```
@keyframesmyAnimation
{
}
```

The defined **@keyframes** does not include any animation at this time.



Note: CSS @ rules are instructions or directives to the CSS parser. They start with the '@' character followed by an identifier. There are many CSS @ rules such as the @keyframes which is discussed in this topic and @ font-face or @media that are discussed later.

To define an animation inside the **@keyframes**, specify the animation start point, using 0% or a **from** value, and an end point, using the 100% or **to** value.

The following are two simple animation definitions, which when applied to an element, will move it to the right by 100 pixels:

A Simple CSS3 Animation

```
@keyframes move1
{
  from { right: 0px; }
  to { right: 100px; }
}
@keyframes move2
{
  0% { right: 0px; }
  100% { right: 100px; }
}
```

To create more advanced animations, specify individual frames by indicating the percentage of the animation process at which they should appear.

The following example from the preceding code shows the addition of more frames between the start and end points.

An Example of Frames

```
@keyframes move1
{
  from { right: 0px; }
  25% { right: 25px; background: blue; }
  50% { right: 50px; background: red; }
  75% { right: 75px; background: blue; }
  to { right: 100px; }
}
@keyframes move2
{
  0% { right: 0px; }
  25% { right: 25px; background: blue; }
  50% { right: 50px; background: red; }
  75% { right: 75px; background: blue; }
  100% { right: 100px; }
}
```

Both of the above **@keyframes** do the same thing. The animation moves the element to the right by 100 pixels, and while moving the element, changes its background first to blue, then red, and then back to blue.

After defining the animation by using **@keyframes**, the animation must be applied to an element. This requires the use of a number of animation properties, such as **animation-name**, **animation-duration**, and the **animation** shortened property. The following terms provide explanation of some of the animation properties:

animation-name. Indicates the name of the **@keyframes** animation to be used. If the name is none, there will be no animation—which should be used to override or stop the animation.

The following example shows how to set an animation-name.

The animation-name property

```
div
{
```

```
animation-name: myAnimation;
}
```

animation-duration. Defines the duration of the animation, that is, how long it will take to run from start to finish. The number provided is evaluated in milliseconds or in seconds, if the value is followed by an s. The default value of the property is 0, so if the property is not used, no animation will be run.

The following example shows how to set an animation-duration.

The animation-duration property

```
/* sets the animation to 5 seconds long */
div
{
  animation-duration: 5s;
}
```

animation. a shortened function that receives all possible animation properties, including the name and duration. The shortened function can receive six parameters: **name**, **duration**, **timing-function**, **delay**, **iteration-count**, and **direction**.



Note: For more information about **timing-function**, **delay**, **iteration-count**, and **direction**, see their specifications at the following link: [CSS3 Animations](#)

The following is an example of using the animation property.

The animation property

```
div
{
  animation: myAnimation 5s linear 3s infinite alternate ;
}
```


Text Effects

- New text properties such as:
 - text-shadow
 - word-wrap
- @font-face property helps to use non-regular fonts

```
h1
{
    text-shadow: 3px 8px 4px #0000ff;
}
@font-face
{
    font-family: myFont;
    src: url('myFont.woff');
}
```

Text Effects

Text Effects

Text Effects

CSS3 adds new text effects properties such as **text-shadow** and **word-wrap**. Also, CSS3 adds the **@font-face** rule into the CSS specifications, helping integrate new and non-standard fonts into the website or application.

text-shadow. Applies shadow to the text of the element. In the past, to create shadowed text, developers mostly used images of text that had a shadow. This property can decrease the number of images on the webpage and improve webpage performance. The text-shadow property receives four parameters:

horizontal shadow, vertical shadow, blur distance, and shadow color.

The following is an example of text-shadow.

Text-shadow

```
h1
{
    text-shadow: 3px 8px 4px #0000ff;
}
```

word-wrap. In the past, if a word was too long to fit the element size, the word simply went over the element boundaries. This means that sometimes long words and sentences had to be manipulated. The **word-wrap** property enables the developer to apply a style that forces the text to wrap, even if that means splitting a word in the middle.

The following is an example of how to apply word-wrap on a paragraph element.

Word-wrap

```
p
{
    word-wrap: break-word;
}
```

@font-face. Before CSS3, there were very limited font options. Most options were the fonts already installed in the user computer. Therefore, fonts like Ariel were very common on websites and applications.

CSS3 adds support for the @font-face rule, enabling the developer to integrate their own fonts onto webpages. To integrate fonts onto webpages, include the new font file into a website or application and use the @font-face rule to enable the user to download the font from the website or application. The @font-face has two main properties: font-family and src. The font-family is the given name for the font. The src is the location from where to download the font from the website or application.

The following is an example of creating a @font-face rule.

@font-face

```
@font-face
{
    font-family: myFont;
    src: url('myFont.woff');
}
```



Note: The woff file extension, used in the preceding example, refers to the web Open Font Format—a special font format for use on webpages.

After you create the @font-face rule, to use the font on text on a website or application, use its family name.

The following example shows you how to use @font-face on all the fonts on a webpage (by using the body tag).

Using a @font-face

```
body
{
    font-family: myFont;
}
```

Media Queries

- Media queries allow the developer to tailor the CSS for each device that is used to view the webpage
- They consist of a media type, and zero or more expressions
- The expressions check for conditions of particular media features
- If the condition is met, a specific style sheet is used

```
div
{
    border: solid 1px black;
}
@media screen and (max-width:400px)
{
    div
    {
        border: none;
    }
}
```

Media queries are new CSS3 rules that you can use to detect media features and respond accordingly. With more people using smart phones and tablets, it is very difficult to create websites and applications that can be presented the same way on all these devices. Media queries can help to tailor the CSS for each device by identifying it by using a specific query, such as resolution width, height, and so on.

A media query consists of a type such as screen, projection, and handheld device. After the media type, a zero or more expressions that check for some conditions are followed. These expressions can include, for example, width, height, device-width, device-height, and resolution. Media queries return true or false for the expressions. If the query returns true, the CSS rule is used.

Media queries can be used in three ways: media attribute in the link element, using the **@import** rule, and the **@media** rule in CSS.

The following example shows how to check if the device has a color screen, and if so, to apply a specific color.css file.

Using a link element

```
<link rel="stylesheet" media="screen and (color)" href="color.css" />
```

The following example is the same check that uses the @import rule.

Using the @import rule

```
@import url(color.css) screen and (color);
```

The main difference between the media attribute and the **@import** rule is where the queries exist. To apply the media query on a specific webpage only, use the media attribute. To put the media query and its rule in a CSS file, use the **@import** rule.

The **@media** rule enables developer to create style blocks that are applied if the query returns true.

The following is an example of a media query that uses the @media rule.

@media Example

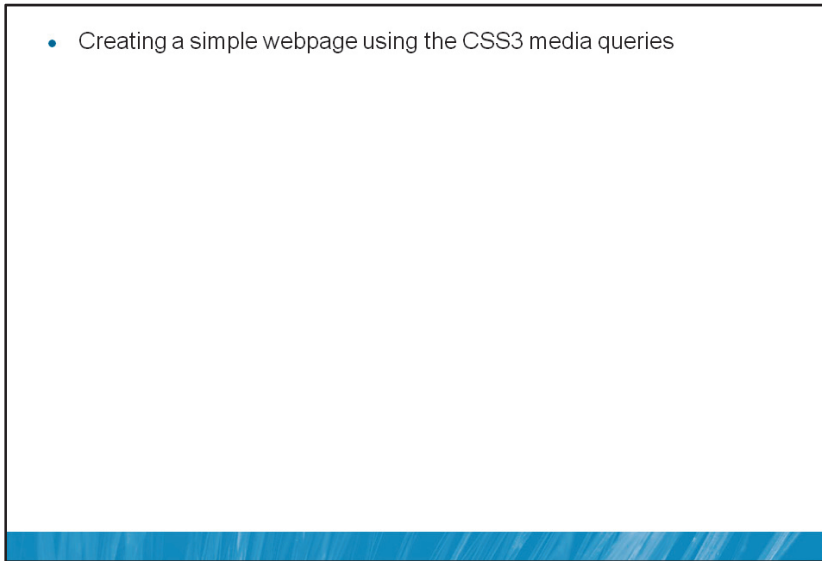
```
div
{
    border: solid 1px black;
}
@media screen and (max-width:400px)
{
    div
    {
        border: none;
    }
}
```

In the preceding example, every div on the webpage has a black border. If the screen is less than 400 pixels, the media query returns true and all of the divs borders are removed. Using such a method can supply a style for each device and improve the site and application usability.



Additional Reading: For more information about media queries, go to the following link:
[CSS3 Media Queries](#)

Demonstration: Media Queries



This demonstration shows how to:

- Create a new Empty Web Application in VS2010
- Add a new HTML file
- Add markup to the web page using CSS3 media queries

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file's body element add the following elements:

```
<div id="element">This element has a border. When the screen's width is less than 400
pixels the border is removed</div>
```

4. In the HTML file's head element add the following style tag:

```
<style type="text/css">
#element
{
    border: solid 1px black;
}
@media screen and (max-width:400px)
{
    #element
    {
        border: none;
    }
}
</style>
```

5. Save the file as an HTML file.
6. Run the saved file.

Lab: Laying Out and Styling Webpages

- Add a CSS3 style to the store's webpages.

Logon Information

Virtual Machine	Virtual machine: SEA-DEV
User Name	User name: Admin
Password	Password: Pa\$\$w0rd

Estimated Time: 45 minutes

Scenario

After examining the current web application, the management of Contoso, Ltd has decided to create a better look and feel for the online store. To do this, they hire a freelance designer. The designer provides new, detailed images for the webpages that improve the look and feel of the online store. Your job is to implement all of the designer's instructions.

Objectives

During the lab you will:

- Create a better look and feel for the online store by using layouts and style.
- Integrate CSS3 features into the webpages.

Exercise 1: Add a CSS3 style to the store's webpages.

Scenario

The designer has sent a woff file with a customized font to integrate into the web application. The designer has also requested to animate the logo by moving it a little, and changing its color from green to white and then back to green. The logo and footer borders must be round and the logo must have a shadow.

The designer supplies the .png files that contain and display all the new styles.

► Task 1: Create the new CSS3 style file.

1. Open the starter solution, which is located in the D:\LabFiles\Starter\M04 folder.
2. Create the **css3.css** file.
3. Add link to the new **css3.css** file on the webpages.

► Task 2: Add the @font-face rule support.

1. Add the @font-face rule with the customized font (**Action_Man-webfont.woff**) located in the **Styles** folder.

► Task 3: Add a box-shadow to the logo and the footer.

1. Add the box-shadow property to the logo and the footer.

► Task 4: Add a Media Query property.

1. Add a media query that checks the max screen width, and if it is below 400 pixels, removes the box-shadow.

► Task 5: Add border-radius support.

1. Add border-radius to the nav element, game-box class, logo, legend, box header, container footer, cart footer, and to all the divs in the fieldset.

► Task 6: Add the nth-child pseudo element.

1. Add the nth-child pseudo element to paint all the odd rows in the cart table with the #e5e5e5 color.

► Task 7: Add multi-columns Support.

1. Add multi-column support to the game-info CSS class.

► Task 8: Add background-size support.

1. Add background-size support to the button class.

► Task 9: Add CSS3 animation.

1. Add animation to animate the logo to change its color from green to white, and to move left and right every 2 seconds.

Results: CSS3 elements are integrated onto the store's webpages.

Module Review and Takeaways

In this module, you learned to lay out webpages, and some key factors in layouts such as white space, fonts, and positioning. This information should help you better communicate with designers and UX experts when discussing and implementing their design layouts. It is your responsibility to take the idea of the layout and implement it on webpages.

You also learned about CSS3 and many of its new capabilities. There is more that you can do with CSS3. Start with what you have learned here, and continue to learn more about it with the links that were supplied in Lesson 2.

Review Questions

What are the basic considerations for creating layouts?

Module 5

Getting Started with Graphics and Multimedia Elements

Contents:

Module Overview	5-2
Lesson 1: Canvas Basics	5-4
Lesson 2: Video/Audio Formats and Codecs	5-51
Lesson 3: Controlling Multimedia with JavaScript	5-71
Lab: Getting Started with Graphics and Multimedia Elements	5-86
Module Review and Takeaways	5-89

Module Overview

- Canvas Basics
- Video/Audio Formats and Codecs
- Controlling Multimedia with JavaScript

Prior to HTML5, the HTML standard had very little innate support for graphics and multimedia. Web developers usually used nonstandard technologies to display graphics or play high-quality video and audio. These technologies were never part of the HTML standard and were implemented as add-ons to the various browsers. Websites relied on the add-on availability for their client market. If the developers responsible for the nonstandard technology did not implement an add-on for a particular browser, visitors using that browser would not get the full experience intended by the website's designers, and the website could easily lose potential customers.

To this day, the wide variety of devices, operating systems, browsers, and form factors continue to pose a difficult challenge for web developers. The various combinations have different levels of support for different graphics and multimedia technologies, and not all of them are supported by add-ons. Even very popular add-ons are not always available. Supporting every device on every platform is complicated and expensive. Web developers have to decide whether to invest in creating multiple tailored websites to target all the different platforms, create fewer websites knowingly excluding potential audiences, or use more primitive technologies found on every platform.

HTML5 aims to solve these problems. The HTML5 standard adds innate support for graphics and multimedia, and makes them easier to work with. Because they are part of the standard, they are supported by a larger number of browsers on more platforms. Web developers can target a larger audience because they can build fewer versions of their website and have a high level of confidence that they will run on more platforms. As browsers become more compliant with the HTML5 standard, the websites will run better on more platforms, with little or no effort on the web developer's part.

This module covers the **canvas** element, which displays graphics, and the **audio** and **video** elements, which are used to play multimedia. These elements represent HTML5's solution to add-on availability and cross-browser compatibility for multimedia and graphics.

Objectives

After completing this module, you will be able to:

- Use the **canvas** element and manipulate it by using JavaScript.
- Use the **audio** and **video** elements, and control them by using JavaScript.
- Add support for multiple audio and video codecs.

Lesson 1

Canvas Basics

- Working with the Canvas
- Drawing Shapes
- Applying Styles
- Drawing Images
- Drawing Text
- Transformations

The web browser is primarily designed to display webpages containing simple content, such as text, lists, links, tables, and images. But the HTML standard also has support for additional kinds of content that can be embedded in a webpage. The embedded content has traditionally been embedded by using the HTML **object** element, which describes the embedded content and provides information from where it can be downloaded. To see and interact with the embedded content, the user installs a browser add-on, also known as an *extension* or *plug-in*, that can parse the **object** element, download the embedded content, and display it inside the browser.

Web developers use the **object** element and other similar mechanisms to display graphics for which the browser does not have built-in support. To create the embedded content, web developers usually cannot use the same tools used to create the website. They have to use different tools and, almost always, languages other than HTML or JavaScript.

The **object** element can still be used to add embedded content, but it has a number of disadvantages:

- If a visitor chooses not to install an add-on, the embedded content that the particular add-on supports cannot be displayed.
- If the add-on developers do not build an add-on for a certain browser, the embedded content cannot be displayed for all the potential visitors with those browsers.
- If a web developer is not familiar with the tool and/or the language required to create the embedded content, the web developer resorts to less attractive technologies.

The new HTML5 **canvas** element is designed to help solve these graphics problems. The canvas is part of the HTML standard and is designed to provide graphics support in browsers without requiring additional add-ons. As soon as any given browser supports the canvas, every visitor with that browser can see the content, so the potential audience can grow without relying on third-party vendors. In addition, the web developer can develop content for the canvas by using the tools they are comfortable with, using nothing more than HTML and JavaScript.

This lesson introduces the HTML5 canvas and illustrates how to use JavaScript to draw simple graphics.

Lesson Objectives

After completing this lesson, you will be able to:

- Start working with a canvas.
- Draw simple and complex shapes on the canvas.
- Apply styles, including colors, gradients, and line settings, to the drawing.
- Draw images and video on the canvas.
- Draw text on the canvas.
- Apply transformations to the canvas and manage the canvas state.

Working with the Canvas

- The **canvas** element is a drawing surface
- The **canvas** element is *lookless*
- The **canvas** element's content is its *fallback content*
- The "2d" context draws two-dimensional drawings on the canvas

```
<canvas id="canvas" width="300" height="150"></canvas>
```

```
var canvas = document.getElementById("canvas");
var context = canvas.getContext("2d");
```

The canvas is fun!
The canvas is fun!

What Is the Canvas?

The canvas is a basic drawing surface. Using JavaScript, the web developer can draw almost anything on the canvas, including charts, graphs, animations, and interactive applications such as games and video players. Although the HTML5 standard defines only a 2D drawing application programming interface (API) at present, it also makes provisions for additional drawing APIs that can be used to draw on the canvas, such as 3D drawing APIs or other APIs that have not yet been designed. This lesson introduces the 2D drawing API defined by the HTML5 standard and supported by most HTML5-compliant browsers.

Different platforms and browsers have different performance characteristics, so the canvas may not perform consistently on all the supporting platforms. However, the canvas design takes performance into consideration, and some browsers use hardware acceleration to optimize performance. Hardware-accelerated browsers can take advantage of the device's Graphics Processing Unit (GPU) to draw graphics faster. Web developers should consider their target audiences before starting canvas development, and test their applications on the target platforms as they develop their applications.

To use a canvas, a **canvas** element is added to the page by using HTML markup. This defines a region on the page for drawing. Then, JavaScript draws on the canvas. HTML markup cannot be used to draw on the canvas.

The canvas is a *lookless* element. By default, it is transparent, so the webpage shows through any part of the canvas that has not been drawn on. It has no borders, margins, or padding, unless they are specified with CSS. The **canvas** element is an inline element, so it flows with the content that surrounds it, unless its layout is modified with CSS. CSS can be used, for example, to change its background with the **background-color** or **background-image** properties, or other background styles.



Note: Not every facet of the **canvas** element can be set with CSS. For example, fonts defined by using CSS are not applied to text drawn on the canvas.

Finally, the canvas represents a fixed-size bitmapped drawing; internally, the canvas contains information about each individual pixel that was drawn on the canvas. The API gives the web developer access to bitmap information. The canvas does not store information about how the different pixels are related to each other. For example, the canvas does not remember the commands that drew each pixel, so it cannot “undo” a command or reapply it automatically when the canvas is resized. The canvas does not have layers, so it does not discern if a shape is “above” or “below” other shapes. The canvas does not store vector graphics or resize the content when the **canvas** element is resized. The canvas does not know about timelines and transitions, so it cannot automatically perform animations. Several JavaScript libraries are available to help perform some of these functions; but ultimately, adding support for these features (undo/redo, vector graphics, layers, animations, and so on) is the web developer’s responsibility.

The canvas Element

The **canvas** element is added to the page by using HTML markup. A canvas has a fixed size, which is defined by using the **width** and **height** attributes. The default width is 300 pixels and the default height is 150 pixels, similar to the following example.



Best Practice: The **width** and **height** attributes are not required. However, the **width** and **height** CSS properties do not affect the canvas the same way they affect other elements. Specifically, the CSS properties do not define the **canvas** element’s size on the page, as illustrated in this training. Specifying the element’s size in HTML or using JavaScript to explicitly set the element’s size is the safest way to define the **canvas** element’s size.

The following example adds a canvas element and sets its id, width, and height attributes.

A Simple Canvas

```
<canvas id="canvas" width="300" height="150"></canvas>
```

The **id** attribute is commonly used because it makes it easy to find the element by using the **document.getElementById** method. However, other methods can be used to retrieve the element from the DOM.

After the canvas is declared, the element can be retrieved by using JavaScript.

Retrieving the canvas Element by Using JavaScript

```
var canvas = document.getElementById("canvas");
```

Fallback Content

The content of the **canvas** element is *fallback content*. Browsers that disable or do not recognize the **canvas** element show the fallback content, instead of the canvas. The fallback content can contain text, an image, or other HTML. Fallback content can be used to inform the user to enable the canvas or download a newer browser. Fallback content can also be used to show an alternate implementation. For example, an interactive graph drawn with the canvas on the browser could alternatively be generated as a static image on the server, if required. Browsers that support the **canvas** element do not display the fallback content.

Any HTML can be added as the fallback content for a canvas element.

Fallback Content for a Canvas Element

```
<canvas id="canvas" width="300" height="150">
```


Please enable the canvas or download a new browser that supports it. You're missing out on some really great graphics!
</canvas>



Best Practice: When using the HTML schema (as opposed to the XHTML schema, which is used in the examples throughout this lesson), some browsers do not require the closing **canvas** tag. Leaving out the closing tag can create an ambiguous DOM structure, because the browser parser may not be able to determine whether the elements appearing after the opening tag are fallback content or not. This can also produce different results on different browsers. We strongly recommend that you always use a closing tag to remove the ambiguity and ensure consistent behavior across browsers.

The 2D Context

The **canvas** element cannot be drawn to directly. The canvas exposes an intermediary that can be used to draw on the canvas. The intermediary is known as a *context*. To retrieve a context, call the **canvas** element's **getContext** method, and pass a string argument identifying the context type. The **getContext** method has the following signature.

```
canvas.getContext(contextId)
```

The **contextId** for the built-in 2D drawing context is "2d".

To retrieve a 2D drawing context, call the **getContext** method with the "2d" argument.

Retrieving a 2D Drawing Context

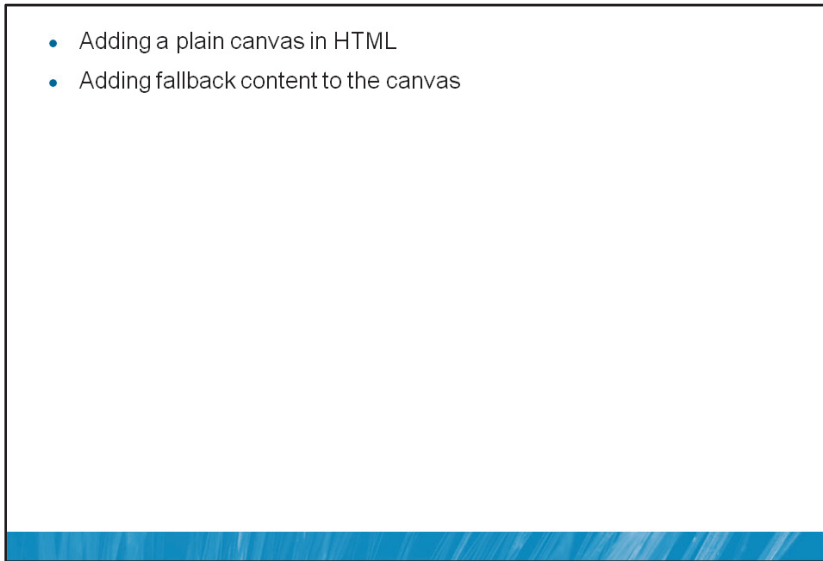
```
var canvas = document.getElementById("canvas");  
var context = canvas.getContext("2d");
```



Best Practice: By design, the canvas may expose different context types. To retrieve a different context, pass the other context's identifier to the **getContext** method. A list of publicly available contexts can be located on a special webpage set aside by the HTML standard committee to publicize contexts and avoid naming conflicts: Contexts

In the remainder of the lesson, the 2D drawing context is referred to simply as "the context".

Demonstration: Canvas with Fallback Content



This demonstration shows how to:

- Add a plain canvas to a webpage.
- Add fallback content to the canvas.
- Verify that fallback content is only visible on browsers that lack support for the canvas.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file, add a **canvas** element with fallback content inside the **body** element.

```
<body>
  <canvas id="canvas">
    <div>Your browser does not support the canvas element.</div>
  </canvas>
</body>
```

4. In the HTML file, add a **style** element inside the **head** element before the **head** element's closing tag.

```
<head>
  ...
  <style type="text/css">
    canvas { border: solid 1px green; }
    div { background-color: orange; }
  </style>
</head>
```

5. Save the file.
6. Run the saved file.

Drawing Shapes

- Simple Shapes: Rectangles
- Complex Shapes: Paths
 - Straight Lines
 - Arcs
 - Bezier Curves
 - Rectangles
 - Clipping Regions

```
context.fillRect(x, y, width, height)
context.strokeRect(x, y, width, height)
context.clearRect(x, y, width, height)
```

The Canvas Grid

The canvas uses a Cartesian coordinate grid system to determine where things are drawn. The upper-left corner is the origin, where x and y are both zero. The x increases towards the right and the y increases towards the lowermost part.

The origin remains the same, regardless of the page direction. For example, if the direction of the body is changed to right-to-left by using CSS (`direction: rtl;`), the canvas itself would be laid out on the page in the right-to-left flow, just like every other element. However, the point of origin in the grid is still the upper-left corner. The notable exception to this rule is text, which is explained later in this lesson.

Drawing Simple Shapes

The context has only one simple shape: the rectangle. The context exposes several methods that can draw a rectangle: **fillRect**, **strokeRect**, and **clearRect**. They have very similar signatures.

```
context.fillRect(x, y, width, height)
context.strokeRect(x, y, width, height)
context.clearRect(x, y, width, height)
```

All three methods accept four arguments: the **x** and **y** coordinates of the rectangle's upper-left corner, and the rectangle's **width** and **height**. The **fillRect** method draws a solid filled-in rectangle. The **strokeRect** method draws only the rectangle's outline. The **clearRect** method erases the bitmap information from the rectangle, so the background is transparent.

The following example draws simple rectangles.

Drawing a Filled-In Rectangle

```
var canvas = document.getElementById("canvas");
var context = canvas.getContext("2d");
context.fillRect(20, 20, 100, 30);
context.strokeRect(20, 60, 100, 30);
context.clearRect(50, 40, 40, 30);
```



FIGURE 5.1: RECTANGLES

The example code draws three rectangles. The third rectangle, drawn with the **clearRect** method, overlaps both the solid rectangle, drawn with the **fillRect** method, and the rectangle outline, drawn with the **strokeRect** method. Because the **clearRect** method erases image information, it removes a portion of the solid rectangle and part of the outline.

Drawing Complex Shapes

Complex shapes are drawn by using the *path*. The path is a list of subpaths that together form a shape. A subpath is a series of points connected by straight or curved lines. Subpaths are not necessarily connected to each other. In general, using a path to draw a shape consists of the following steps:

1. Starting a new path
2. Adding subpaths with straight and curved lines to the path
3. Drawing the whole path at once

Points, lines, and shapes are not rendered until the whole path is drawn. The context can only have one current path, so only one shape can be drawn at a time. To draw multiple shapes, the process is repeated for each shape.

The methods affecting the path have different purposes:

- Some methods add lines to the current subpath. These methods describe the line's type and end point. The methods connect the previous line to the new end point, so they do not specify the start point. Their names usually end with "To", as in "moveTo", "lineTo", and "arcTo".
- Some methods add closed figures to the current subpath. Their starting and end points are identical.
- Other methods provide instructions affecting how subsequent lines are drawn.

Starting a New Path

The **beginPath** method starts a new path and indicates that a new shape is being drawn. The **beginPath** method has the following signature.

```
context.beginPath()
```

The method has no parameters. If a path is already started and has some subpaths, the path is first emptied and the existing subpaths discarded. Any straight or curved lines that are cleared from the path are not drawn.

To start a path, call the `beginPath` method.

Calling the `beginPath` Method

```
context.beginPath();
```

Adding Subpaths

Before anything can be drawn, the position of the first point to be drawn must be set. The **`moveTo`** method creates a new subpath and sets the first start point's position. The next straight or curved line is added to the subpath created by the **`moveTo`** method. The **`moveTo`** method has the following signature.

```
context.moveTo(x, y)
```

The method has two parameters, **`x`** and **`y`**, representing the new point's coordinates. The next straight or curved line is drawn from that point. The **`moveTo`** method itself does not actually draw anything.

The `moveTo` method creates a new subpath at the specified coordinates.

Calling the `moveTo` Method

```
context.moveTo(30, 20);
```

In the example, a new subpath is created and added to the path. The next line added to the subpath is drawn from the point at (30, 20).



Best Practice: Using the **`moveTo`** method, subpaths that are not connected to each other can be added in a single path. Programmers often take advantage of this behavior to draw multiple shapes at the same time, instead of splitting them into multiple paths. This should usually be avoided. Drawing multiple shapes by using one path usually indicates a poor separation of responsibilities in the code. For example, if the canvas is used to draw a bar chart, it is inadvisable to draw all the bars by using one path. Each bar should be drawn by using its own path. A path is easiest to work with when it represents a single independent shape. There are instances though where this technique is warranted. For example, if the bars are filled by using a special pattern consisting of disconnected parallel lines, it makes sense to use a single path for all the lines in each bar. In this case, it would be necessary to use **`moveTo`** to add multiple disconnected subpaths to the path.

Straight Lines

The **`lineTo`** method adds a straight line to the path. It has the following signature.

```
context.lineTo(x, y)
```

Its two parameters, **`x`** and **`y`**, are the coordinates of a new point to add to the subpath. A straight connecting line is drawn between the last point on the subpath and the new point.

The `lineTo` method adds a line ending at the specified coordinates to the subpath.

Calling the `lineTo` Method

```
context.lineTo(70, 20);
```

This code adds a new point to the subpath at the coordinates (70, 20) and draws a straight line from the previous point on the subpath to the new point.



Best Practice: If no points are added to the subpath before the `lineTo` method is called, it behaves like the `moveTo` method. In other words, it sets the start point for the next line to the specified coordinates and does not actually draw a line. To avoid unexpected behavior, always ensure that the `lineTo` method is called after at least one point is added to the subpath. We recommend that you always call the `moveTo` method immediately after the `beginPath` method to ensure that the first subpath has at least one point.

Drawing Paths

The **fill** and **stroke** methods draw the current path on the canvas. The two methods have similar signatures.

```
context.fill()
context.stroke()
```

Both methods have no parameters. Like the **fillRect** and **strokeRect** methods, the **fill** method draws a solid shape and the **stroke** method draws only an outline.

If the shape is open, the **fill** method draws it as though it is closed, but does not actually close the shape. For example, if there are only two connected lines in a subpath when the **fill** method is called, a solid triangle is drawn, even though the third line is missing. The third line is not added to the path.

The **stroke** method does not close the path. Only lines actually described on the path are drawn.

The **fill** and **stroke** methods do not modify the path, do not add lines or subpaths, and do not delete lines or subpaths.

This can be useful when a shape must be filled in and outlined, as the following example shows.

Using the fill and stroke Methods

```
context.beginPath();
context.moveTo(20, 20);
context.lineTo(80, 20);
context.lineTo(80, 60);
context.lineTo(20, 60);
context.lineTo(20, 20);
context.fillStyle = "orange";
context.fill();
context.stroke();
```



FIGURE 5.2: FILLED RECTANGLE

The example code sets the **fillStyle** property, which has not yet been introduced. For now, it is enough to note that it changes the rectangle's color. The important thing to note is that the same path is drawn

twice, once with the **fill** method and then with the **stroke** method. To clear the path or create a new path, call the **beginPath** method.

Closing Subpaths

The **closePath** method closes the figure described by the current subpath, ends the subpath, and creates a new subpath. Closing a shape requires drawing a straight line connecting the first and last points in the subpath. The method has the following signature.

```
context.closePath()
```

The **closePath** method has no parameters.

The following example creates a triangle by using the **closePath** method.

Closing a Triangle

```
context.moveTo(30, 30); // set the first point
context.lineTo(60, 90); // draw a line from (30, 30) to (60, 90)
context.lineTo(90, 30); // draw a line from (60, 90) to 90, 30)
context.closePath(); // draw a line from (90, 30) to (30, 30) to close the triangle
// context.lineTo(45, 60);
```

When there are multiple lines on the subpath, the **closePath** method is equivalent to closing a figure by using the **lineTo** method, and then creating a new subpath by using the **moveTo** method, where both methods are called with the same coordinates. The start point for the new subpath is the same as the previous subpath's start point. In the example, the commented line at the end draws a line from (30, 30) to (45, 60) because the **closePath** method sets the start point to (30, 30).

Arcs

An arc is part of a circle's circumference. There are two ways to draw arcs on the canvas. The first uses the **arc** method, which has the following signature.

```
context.arc(x, y, radius, startAngle, endAngle, anticlockwise)
```

The circle's center is the point at (**x**, **y**) and its radius is the **radius** argument value. The **startAngle** and **endAngle** arguments specify how far in radians the arc's start and end points are from the x-axis. They are measured from the x-axis to the right of the circle's center.

Between any two points on the circle's circumference, there are two arcs, and the **anticlockwise** argument specifies which of the two arcs to draw. When **anticlockwise** is false, the **arc** method adds the arc that follows the clockwise direction from the start point to the end point; otherwise the complementary arc is added. The **anticlockwise** argument is optional and its default value is **false**.

The following example draws two arcs in opposite directions.

Drawing Arcs by Using the arc Method

```
context.beginPath();
context.arc(70, 75, 50, 0, 0.5 * Math.PI);
context.stroke();
context.beginPath();
context.arc(230, 75, 50, 0, 0.5 * Math.PI, true);
context.stroke();
```

The example draws two nearly identical arcs. The only difference is that the first arc (the one on the left) selects the arc in the clockwise direction, and the second selects the complementary arc in the counterclockwise direction.



FIGURE 5.3: ARCS

Note the use of the `Math.PI` constant. Radians are most conveniently expressed in terms of π . A full circle is 2π radians long. A quarter of a circle is 0.5π radians long, which is why the first arc is as long as a quarter of the circle's circumference. The formula for converting degrees to radians is: $\text{radians} = \text{degrees} * \pi / 180$. So, a semi-circle with 180 degrees is an arc with a length of π .

The circle's center is used as a reference point and is not drawn or added to the subpath. Both the start point and the end point are added to the subpath. If the subpath already has points on it, a straight line is drawn connecting the last point on the subpath to the arc's start point.

The following arc is automatically connected to the subpath.

An Implicitly Connected Arc

```
context.beginPath();
context.moveTo(20, 20);
context.arc(70, 75, 50, 0, Math.PI);
context.stroke();
```

The straight diagonal line in the image is added implicitly because the subpath already has a point at (20, 20). If the **fill** method is called and the path contains an open arc, the arc is implicitly closed with a straight line from its start point to its end point.

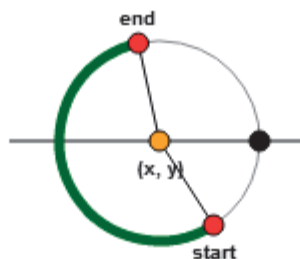


FIGURE 5.5: XY CIRCLE

The second method that can draw an arc is the **arcTo** method, which draws arcs forming rounded corners. The method has the following signature.


```
context.arcTo(x1, y1, x2, y2, radius)
```

The two points at (**x1**, **y1**) and (**x2**, **y2**) are used as control points, or reference points, to calculate the arc. The point (x1, y1) represents the corner before it is rounded. It is where two *imaginary* lines meet:

- The first imaginary line has the last point on the current subpath, which is referred to as (x0, y0), and ends at the corner.
- The second imaginary line has the point (x2, y2) and also ends at the corner.

Between the two imaginary lines is an imaginary circle. The circle touches each line just once along its circumference. The circle's radius is the **arcTo** method's **radius** argument, and also the distance between each imaginary line and the circle's center. The two points where the circle touches the imaginary lines are the *tangential points*.

Once all of these points are calculated, the arc is drawn in two stages:

1. First, a straight line is drawn from (x0, y0) to where it meets the imaginary circle at the first tangential point.
2. Then the shorter arc connecting the tangential points is drawn.

Only the tangential points are added to the subpath; all the other points are only used to facilitate the calculations.

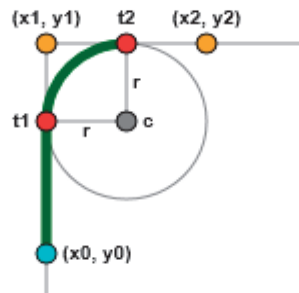


FIGURE 5.6: XY ARC

This image helps visualize the **arcTo** method's behavior. The two gray imaginary lines intersect at (x1, y1), the orange point in the corner. The blue point at (x0, y0) is the last point on the current subpath before the arc is drawn, so the **arcTo** method starts there. The first green segment is a straight line drawn from (x0, y0) to t1, the first tangential point. The second green segment is the arc itself. It is drawn along the gray imaginary circle's circumference all the way to t2, the second tangential point. The circle's radius, specified by the **radius** argument, specifies how far the circle's center is from both imaginary lines. After the arc is drawn, the subpath has two new points, at t1 and t2. The point t2 becomes the last point on the current subpath.

The following example draws a rectangle with rounded corners.

Drawing Rounded Corners by Using the arcTo Method

```
context.beginPath(); // start a new subpath for the rectangle
context.moveTo(150, 30); // set the first point in the subpath
context.arcTo(270, 30, 270, 75, 20); // draws an arc with a 20-pixel radius for the top
right corner
```

```

context.arcTo(270, 120, 150, 120, 20); // draws an arc with a 20-pixel radius for the
bottom right corner
context.arcTo(30, 120, 30, 75, 20); // draws an arc with a 20-pixel radius for the bottom
left corner
context.arcTo(30, 30, 150, 30, 20); // draws an arc with a 20-pixel radius for the top
left corner
context.lineTo(150, 30); // closes the rectangle
context.fill(); // draws the solid filled-in rectangle with rounded corners

```



FIGURE 5.7: ROUNDED CORNERS

A zero length **radius** draws a simple pointed corner with no arc. The **radius** argument must not be less than zero. If all three points—(x0, y0), (x1, y1), and (x2, y2)—are on the same line, a simple straight line is drawn to the corner point (x1, y1), ignoring the second reference point (x2, y2).

Quadratic Bezier Curves

Another type of curved line supported by the canvas is the quadratic Bezier curve, which is drawn by using the **quadraticCurveTo** method. The method has the following signature.

```

context.quadraticCurveTo(cpx, cpy, x, y)

```

The first point (**cpx, cpy**) is the control point and the end point is (**x, y**). Like the arc, the control point is where two imaginary lines meet. However, the curve connecting the two lines follows a more complex parabolic path, instead of the arc of an imaginary circle. The curved line is drawn from the last point on the current subpath to the point at (x, y), which is added to the subpath. A convenient way to visualize it is that the control point “pulls” the curve toward it.

The following example draws a quadratic Bezier curve.

Drawing a Quadratic Bezier Curve

```

context.beginPath();
context.moveTo(30, 120);
context.quadraticCurveTo(270, -40, 220, 120);
context.stroke();

```



FIGURE 5.8: QUADRATIC BEZIER CURVE

If no points are added to the current subpath before the **quadraticCurveTo** method is called, a straight line is drawn from the control point (cpX, cpy) to the end point (x, y) and both points are added to the subpath.

Cubic Bezier Curves

The **bezierCurveTo** method adds a cubic Bezier curve to the path. The method has the following signature.

```
context.cubicCurveTo(cp1x, cp1y, cp2x, cp2y, x, y)
```

The first two points (**cp1x, cp1y**) and (**cp2x, cp2y**) are the control points, and (**x, y**) is the end point. The cubic Bezier curve is very similar to the quadratic Bezier curve, except that it uses another control point. Each control point “pulls” the line toward itself. Instead of two imaginary lines, the cubic Bezier curve can be thought of as having three imaginary lines:

- The first imaginary line connects the last point on the current subpath (x0, y0) to the first control point (cp1x, cp1y).
- The second imaginary line connects the two control points.
- The third imaginary line connects the second control point (cp2x, cp2y) to the end point (x, y).

The following example draws a cubic Bezier curve.

Drawing a Cubic Bezier Curve

```
context.beginPath();
context.moveTo(30, 75);
context.bezierCurveTo(70, 20, 160, 130, 220, 75);
context.stroke();
```

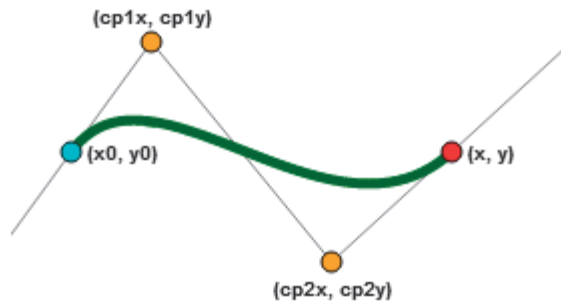


FIGURE 5.9: CUBIC BEZIER CURVE

If no points are added to the subpath before the **bezierCurveTo** method is called, the first control point (cp1x, cp1y) is added to the subpath as the start point and a cubic Bezier curve is drawn to the end point by using both control points. In other words, the first control point is used twice.

Rectangles

The methods to draw rectangles, introduced earlier, are not added to the path and do not affect the path. The methods are independent and can be drawn at any time, even while lines are being added to the current path. In contrast, the **rect** method adds rectangles to the path. The **rect** method's signature is similar to its "simple" counterparts.

```
context.rect(x, y, width, height)
```

The **rect** method creates a new subpath, adds all of the rectangle's four points to the new subpath, and closes the subpath. The **rect** method then creates a new subpath and adds the start point (**x**, **y**) to the second subpath. It does not automatically connect any previous subpaths to the rectangle's subpath.

The following example demonstrates the **rect** method's behavior.

Using the rect Method

```
context.beginPath();
context.moveTo(30, 30);
context.lineTo(120, 30);
context.rect(40, 40, 70, 40);
context.lineTo(75, 60);
context.stroke();
```



FIGURE 5.10: RECTANGLE

Three subpaths are created in this drawing. The first subpath is created explicitly when the **beginPath** method is called. When the **rect** method is called, a new subpath, disconnected from the first, is implicitly created for the rectangle. The **rect** method also closes the rectangle's subpath and creates a third

subpath, adding to it the rectangle's start point. The **lineTo** method that draws the diagonal line that is added to the third subpath.

The Clipping Region

Clipping is a technique that masks part of a drawing to limit the areas that can be drawn on. In other words, when a clipping region is used, parts of the canvas are "clipped" so that even if something is drawn there, it is not rendered and does not actually apply to the drawing. It is similar to using spray paint with cutout sheets: the spray may fly in any direction, but only the exposed surfaces actually get painted.

After a path is created, the **clip** method can be used to create a clipping region with that path. The **clip** method has the following signature.

```
context.clip()
```

When the **clip** method is called, the path is used as the clipping region. Anything drawn on the canvas after the clipping region is created is constrained to the area inside the path. If the path is not closed, it is closed implicitly.

To use a clipping region, perform the following steps:

1. Create a path.
2. Call the **clip** method.
3. Draw on the canvas.

The following example creates a clipping region.

Creating a Clipping Region

```
context.beginPath()
context.arc(150, 75, 60, 0, 2 * Math.PI);
context.clip();
for(var i = 0; i < canvas.height; i++) {
  var red = Math.floor(Math.random() * 255);
  var green = Math.floor(Math.random() * 255);
  var blue = Math.floor(Math.random() * 255);
  context.beginPath();
  context.moveTo(0, i);
  context.strokeStyle = "rgb(" + [red, green, blue].join(",") + ")";
  context.lineTo(canvas.width, i);
  context.stroke();
}
```

The code in the example paints a circle by using lines with random colors. Writing code to paint just the circle is more difficult than using a clipping region. So, instead of drawing the circle, the circle is added to the path by using the **arc** method, and everything around it is clipped by using the **clip** method. The loop draws randomly colored lines horizontally across the whole canvas width. Because of the clipping region, only the circle is painted.

Because the colors are random, every time the script runs, a different circle is produced. This is one of those circles.



FIGURE 5.11: CIRCLE CLIPPING REGION

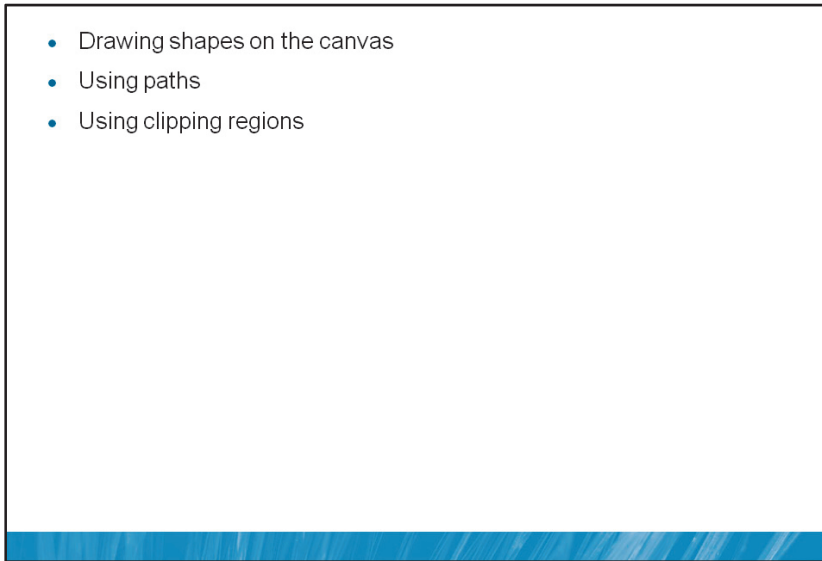
The following example resets the clipping region so the whole canvas can be drawn on again.

Resetting the Clipping Region

```
context.beginPath();  
context.rect(0, 0, canvas.width, canvas.height);  
context.clip();
```

This code creates a path, adds a rectangle covering the whole canvas, and then calls the **clip** method to set the clipping region to the whole canvas. This whole canvas is also the default clipping region.

Demonstration: Drawing Shapes



This demonstration shows how to:

- Draw a rectangle.
- Draw a circle.
- Use a path and a clipping region.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file, add a **canvas** element inside the **body** element.

```
<canvas id="canvas">
</canvas>
```

4. In the HTML file, add a **script** element inside the **head** element before the **head** element's closing tag.

```
<head>
...
<script type="text/javascript">
</script>
</head>
```

5. Inside the **script** element, call the new **draw** function after the page finishes loading, and retrieve the canvas context before starting to draw.

```
<script type="text/javascript">
function draw() {
    var canvas = document.getElementById("canvas");
    var context = canvas.getContext("2d");
}
```

```
document.addEventListener("DOMContentLoaded", draw, false);
```

</script>

- At the end of the **draw** function, add the following code to draw a filled rectangle.

```
function draw() {  
    ...  
    var context = canvas.getContext("2d");  
    context.fillRect(20, 30, 100, 40);  
}
```

- Save the HTML file.
- Run the saved file.
- At the end of the **draw** function, add the following code to draw the circumference of a circle.

```
function draw() {  
    ...  
    context.arc(70, 110, 30, 0, 2 * Math.PI, true);  
    context.stroke();  
}
```

- Save the HTML file.
- Run the saved file.
- At the end of the **draw** function, add the following code to draw part of a rectangle inside a clipping region defined by a circle.

```
function draw() {  
    ...  
    context.beginPath();  
    context.moveTo(260, 75);  
    context.arc(230, 75, 30, 0, 2 * Math.PI, true);  
    context.clip();  
    context.beginPath();  
    context.rect(180, 55, 100, 40);  
    context.fill();  
}
```

- Save the HTML file.
- Run the saved file.

Applying Styles

- Colors
- Gradients
 - Linear Gradients
 - Radial Gradients
- Line Settings

butt

round

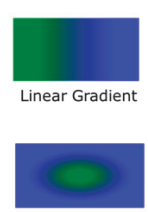
square

Line Settings



Linear Gradient

Radial Gradients



```
context.fillStyle = style
context.strokeStyle = style

context.createLinearGradient(x0, y0, x1, y1)
gradient.addColorStop(0.25, "red");
gradient.addColorStop(0.5, "green");
context.fillStyle = gradient;
```

Colors

The context exposes two properties by specifying the styles used to paint any shape on the canvas: **fillStyle** and **strokeStyle**. Both properties have the same signature.

```
context.fillStyle = style
context.strokeStyle = style
```

The **fillStyle** is used when any filling method is called (for example, **fill**, **fillRect**) and the **strokeStyle** is used when any stroking method is called (for example **stroke**, **strokeRect**).

The simplest style is a color. A color can be specified with the CSS syntax for colors. A color can be provided in any of the following formats, and in other formats permitted by the CSS standard:

- A color name string, such as "red" or "gainsboro"
- A hex value, such as "#aa33cc"
- An RGB value, such as "rgb(50, 120, 200)"
- An RGBA value supporting alpha transparency, such as "rgba(200, 20, 100, 0.5)"

To set the color, assign the color to the fillStyle or strokeStyle property before filling or stroking the shape.

Using the fillStyle and strokeStyle Properties

```
context.fillStyle = "green";
context.fillRect(20, 20, 80, 40);
context.fillStyle = "#b31183";
context.fillRect(120, 20, 80, 40);
context.strokeStyle = "rgb(30, 100, 170)";
context.strokeRect(20, 80, 80, 40);
context.fillStyle = "orange";
context.strokeStyle = "black";
context.fillRect(120, 80, 80, 40);
context.strokeRect(120, 80, 80, 40);
context.fillStyle = "rgba(80, 0, 0, 0.5)";
```

```
context.fillStyle = "rgba(0, 80, 0, 0.3)";
context.fillRect(60, 40, 100, 60);
context.strokeRect(60, 40, 100, 60);
```

This code draws five rectangles, where the fifth rectangle overlaps the other four rectangles. Each rectangle specifies the **strokeStyle** and **fillStyle** a bit differently. The large overlapping rectangle in the middle uses partially transparent colors.



FIGURE 5.12: FILLED OVERLAPPING RECTANGLES

Gradients

Styles can also be used to specify gradients. A gradient is a smooth transition from one color to another. A gradient can be used to create many different types of effects and are very popular when working with graphics.

There are three steps to working with a gradient:

1. Create a gradient.
2. Add color stops to the gradient.
3. Assign the gradient to a style.

Color Stops

The canvas supports two gradient types: *linear gradients* and *radial gradients*. When the shape with the gradient is drawn, the canvas creates smooth transitions between each color stop. Color stops are added to both gradients in the same way. After the gradient is created and stored in a variable, the **addColorStop** method can be used to add color stops to the gradient. The **addColorStop** method has the following signature.

```
gradient.addColorStop(offset, color)
```

The **offset** argument specifies the color stop's position along the gradient. The position can be a number between 0 and 1. Adding a color stop at 0.5, for example, places the color stop right in the middle of the gradient. The **color** argument is any valid CSS color.

If no color stops are added, the gradient does nothing. Adding just one color stop is not enough to create a transition, so the whole shape is painted with the color stop's color. To create gradients with real transitions, add two or more color stops.

Color stops can be added in any order. In most cases, the offsets determine their positions along the gradient, not the order in which they were added. The exception to this rule is when multiple color stops are added at the same offset. In that case, the color stops at the same position are applied in the order in which they were added. If more than two color stops are at the same position, the color stops between the first and last color stops are effectively ignored.

Linear Gradients

The simplest gradient is a linear gradient. A linear gradient is described by an imaginary line defining the gradient's length and direction. To create a linear gradient, call the **createLinearGradient** method. The method has the following signature.

```
context.createLinearGradient(x0, y0, x1, y1) → gradient
```

When the gradient is rendered, the transitions start at (**x0**, **y0**), progresses through each color stop, and finishes at (**x1**, **y1**). The method creates and returns a linear gradient object that can be stored in a variable and assigned to the **fillStyle** or **strokeStyle** properties.

The following example draws a linear gradient with three color stops.

Using a Simple Linear Gradient

```
var gradient = context.createLinearGradient(20, 20, 120, 70);
gradient.addColorStop(0.25, "red");
gradient.addColorStop(0.5, "green");
gradient.addColorStop(0.75, "blue");
context.fillStyle = gradient;
context.fillRect(20, 20, 100, 50);
```

This code creates a very simple gradient following the diagonal from the rectangle's upper-left corner to the lower-right corner. The gradient's length in this example is the exact length of the rectangle's diagonal. Color stops for red, green and blue are added at 25%, 50% and 75% of the gradient's length, respectively. Finally, the gradient is assigned to the **fillStyle** property and the rectangle is drawn.



FIGURE 5.13: LINEAR GRADIENT

The linear gradient can also be applied to more complex shapes. The linear **gradient** can have any length; it is not required to be as long as the shape it renders, and it can be shared with multiple shapes.

The following example applies a gradient to more complex shapes.

Using a Linear Gradient with Complex Shapes

```
var gradient = context.createLinearGradient(0, 0, 300, 150);
gradient.addColorStop(0, "orange");
gradient.addColorStop(1, "purple");
context.fillStyle = gradient;
context.beginPath();
context.arc(50, 75, 45, 0.5 * pi, 1.5 * pi);
context.fill();
context.beginPath();
context.rect(50, 30, 100, 90);
context.fill();
context.beginPath();
context.arc(150, 75, 45, 0.5 * pi, 1.5 * pi, true);
context.fill();
```

The example creates a very simple linear gradient describing a transition between two color stops at the gradient's extreme ends, from orange on the left to purple on the right. It then creates a path with a semi-

circle attached to each side of the rectangle. Each segment is drawn with a separate path and shares the same gradient, so the whole connected shape is rendered with one smooth transition.



FIGURE 5.14: LINEAR GRADIENT

Radial Gradients

The radial gradient creates a transition between two different circles instead of points. The **createRadialGradient** method creates a radial gradient and has the following signature.

```
context.createRadialGradient(x0, y0, r0, x1, y1, r1) → gradient
```

The first three arguments describe a circle whose center point is at (**x0**, **y0**) and whose radius is **r0**. The last three arguments describe a circle with the center point at (**x1**, **y1**) and whose radius is **r1**. The gradient follows the cone created by these two circles, from the first circle to the second.

In the simplest radial gradient, the two circles have the same center point, but different radii. If the circles are identical, no gradient can be drawn.

The following example creates a simple radial gradient around a single point.

Using a Simple Radial Gradient

```
var gradient = context.createRadialGradient(75, 75, 10, 75, 75, 60);
gradient.addColorStop(0, "yellow");
gradient.addColorStop(1, "blue");
context.fillStyle = gradient;
context.fillRect(20, 20, 110, 110);
```

The example creates a radial gradient with circle centers in the rectangle center. The first circle has a 10 pixel radius and the second circle has a 60 pixel radius. When the rectangle is drawn, the gradient extends from the inner circle (with the smaller radius) to the outer circle. Because no other colors are defined, the remainder of the rectangle is filled with the color of the last color stop.

The image shows the rectangle produced by the code example on the left and a mirror image on the right. The second rectangle's colors are reversed by switching the radii between the two circles. The first circle has a 60 pixel radius and the second a 10 pixel radius. This is equivalent to switching the positions of the color stops.

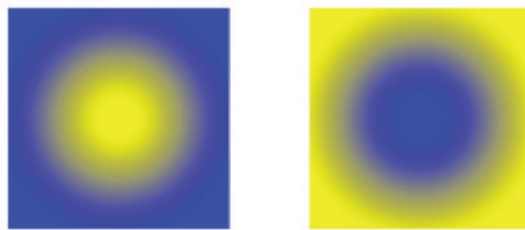


FIGURE 5.15: RADIAL GRADIENT

The circles in a radial gradient do not need to have the same center.

The following example creates a radial gradient with two very different circles.

Using a Complex Radial Gradient

```
var gradient = context.createRadialGradient(50, 75, 10, 205, 75, 80);
gradient.addColorStop(0, "orange");
gradient.addColorStop(0.5, "blue");
gradient.addColorStop(1, "red");
context.fillStyle = gradient;
context.fillRect(20, 20, 260, 110);
```

The example creates a single radial gradient with two circles located at different coordinates and having different radii. The gradient also has three color stops to better visualize the transition from the tiny orange circle on the left to the large red circle on the right.

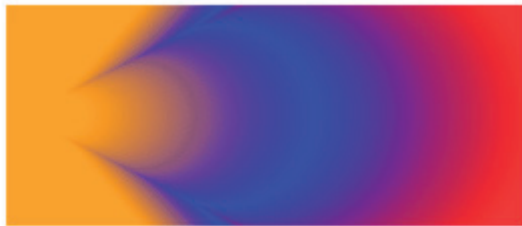


FIGURE 5.16: COMPLEX RADIAL GRADIENT

Gradients can also be applied to the **strokeStyle** to create fancy outlines.

The following code example is identical to the previous example, except that the gradient is applied to the **strokeStyle**, and the **strokeRect** method is used to create the outline.

Using a Complex Radial Gradient for an Outline

```
var gradient = context.createRadialGradient(50, 75, 10, 205, 75, 80);
gradient.addColorStop(0, "orange");
gradient.addColorStop(0.5, "blue");
gradient.addColorStop(1, "red");
context.strokeStyle = gradient;
context.strokeRect(20, 20, 260, 110);
```

This produces the following image.



FIGURE 5.17: COMPLEX RADIAL GRADIENT OUTLINE

Lines

Line Width

The 2d context has a number of properties controlling how lines are drawn. The first is the **lineWidth** property, which controls the width of the line. The **lineWidth** property has a simple signature.

```
context.lineWidth = value
```

The value is a number representing the requested line width. By default, the line width is measured in pixels. If the canvas is scaled, the line width is scaled with the canvas. (Scaling the canvas is covered in a later topic.) The default value is 1.

The value must be a finite positive number greater than zero. If an invalid value is assigned to the property, the value is ignored.

Line Endings

When lines are drawn, the canvas can draw a few different line ending types. The line endings are controlled with the **lineCap** property, which has the following signature.

```
context.lineCap = value
```

Its value can only be one of the following three values:

- "butt"—the end of the line is flat.
- "round"—a semicircle is added to the end of the line.
- "square"—half of a square is added to the end of the line.

Any other value assigned to the property is ignored. The default value is "butt".

The following figure illustrates what each line ending looks like. All three lines have exactly the same length.

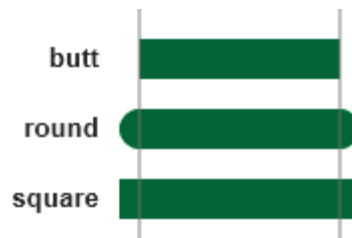


FIGURE 5.18: LINE ENDINGS

The following example shows how to draw the line with the round ends displayed above.

Using the lineCap Property

```
context.beginPath();
context.moveTo(100, 70);
context.lineTo(200, 70); // add the line to the path
context.lineCap = "round"; // set the line ending
context.lineWidth = 20; // make the line very wide to make the lineCap easily observable
context.strokeStyle = "darkgreen";
context.stroke();
```

Joining Lines

When two lines are connected, the canvas provides a few options on how to draw the meeting point between the lines. This may not be very useful for lines only one pixel wide, but it can be very helpful with wider lines. Corners and angles can be created to look sharper or smoother to improve the drawing. The **lineJoin** property determines how to draw the meeting point between two lines in a subpath. The property has the following signature.

```
context.lineJoin = value
```

The **value** argument can accept one of only three values:

- “bevel”—when two lines meet, the corner is “shaved off” on the outside.
- “round”—the corner is rounded off on the outside.
- “miter”—the two lines extend as far as they can until their outside edges meet.

The default value is “miter”. The following figure illustrates what each value looks like. In all three cases, the two lines that form a corner are part of a single subpath.



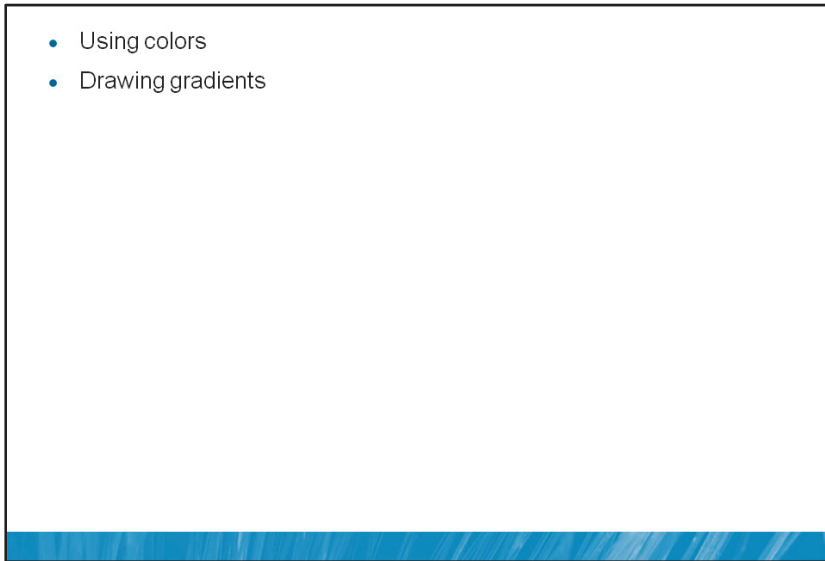
FIGURE 5.19: CORNERS

When the “miter” value is selected, the **miterLimit** property provides finer control over how the corner is drawn at very acute angles. For example, if the last two lines in the previous illustration are closer to each other, the angle becomes more acute and the outer edges would extend much farther out. To avoid this, the **miterLimit** property determines the maximum ratio between the triangle’s length and its base. If the triangle gets too long, it is not drawn at all, although the **lineJoin** was set to “bevel”. The **miterLimit** property has the following signature.

```
context.miterLimit = value
```

The **value** argument can be any positive finite number greater than zero. Its default value is 10.

Demonstration: Applying Styles



- Using colors
- Drawing gradients

This demonstration shows how to:

- Specify colors.
- Draw a gradient with color stops.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file, add a **canvas** element inside the **body** element.

```
<canvas id="canvas">
</canvas>
```

4. In the HTML file, add a **script** element with a **draw** function inside the **head** element, before the **head** element's closing tag.

```
<head>
...
<script type="text/javascript">
  function draw() {
    var canvas = document.getElementById("canvas");
    var context = canvas.getContext("2d");
  }
  document.addEventListener("DOMContentLoaded", draw, false);
</script>
</head>
```

5. At the end of the **draw** function, draw a rectangle with a radial gradient and three color stops.

```
<script type="text/javascript">
  function draw() {
    var canvas = document.getElementById("canvas");
    var context = canvas.getContext("2d");
```



```
var gradient = context.createLinearGradient(0, 0, 300, 150);
gradient.addColorStop(0.1, "blue");
gradient.addColorStop(0.4, "green");
gradient.addColorStop(0.7, "yellow");
context.fillStyle = gradient;
context.beginPath();
context.arc(50, 75, 45, 0.5 * Math.PI, 1.5 * Math.PI);
context.fill();
context.beginPath();
context.rect(50, 30, 200, 90);
context.fill();
context.beginPath();
context.arc(250, 75, 45, 0.5 * Math.PI, 1.5 * Math.PI, true);
context.fill();
}
document.addEventListener("DOMContentLoaded", draw, false);
</script>
```

6. Save the HTML file.
7. Run the saved file.

Drawing Images

- Images can be drawn from various sources:
 - **img** elements
 - **video** elements
 - **canvas** elements
- The **drawImage** overloads can draw, scale, and slice images.
- Patterns of repeated images can be used as stroke and fill styles.



```
var image = document.createElement("img");
image.onload = function () {
    context.drawImage(image, 20, 20);
};
image.src = "image.png";
```

In addition to dynamically-generated graphics, various kinds of media can be drawn on the canvas. The canvas can draw images, video, and canvases on its surface. Integrating media components with regular graphics can help produce very appealing drawings.

Drawing Images Directly to the Canvas

The **drawImage** method draws media on the canvas. The canvas draws bitmaps, essentially two-dimensional colored pixel arrays. It has no way to represent changes over time, so when an animated or script-controlled media source, such as a video, an animated image, or another canvas, is drawn, only a single bitmap—a “snapshot”—can be captured and rendered on its surface.



Note: The different media types are collectively referred to as *images*, including video and canvases. Wherever the word **image** appears in a method or argument name, it refers to any supported media type. In this lesson, unless otherwise specified, the term “image” is used in the same way.

In all the following techniques for drawing various image types, only one “frame” from the source is drawn on the canvas:

- For animated images, it is always the first frame.
- For videos, it is always the current frame.
- For canvases, it is always the canvas bitmap at the moment it is captured.

The canvas does not automatically redraw itself when the source animation changes. This topic is covered more comprehensively in the lesson on canvas animation.

Images drawn on the canvas do not affect the path.



Best Practice: Anytime a canvas tries to access a resource that has to be downloaded, there is potential for misuse. **img** elements are known for violating user privacy because their **src**

attribute is easy to use to send private information in a secretive fashion to a remote party. The **video** element is newer than the **img** element, but shares some of the syntax characteristics and can be similarly misused. Because they can both be used with the canvas, the canvas is also susceptible to misuse.

To protect users, browsers apply certain security restrictions when **img**, **video**, and other canvas elements use resources downloaded from other domains.

To avoid unexpected behavior, use only resources that can be downloaded from the same domain as the canvas.

Drawing a Complete Image

There are a few different versions of the **drawImage** method. Each version, known as an *overload*, receives different arguments and draws an image in a slightly different way. Each overload is more capable than the previous overload. The first and simplest overload has the following signature.

```
context.drawImage(image, dx, dy)
```

The **image** argument can be an **img**, **video**, or **canvas** element. The **dx** and **dy** arguments specify the point at which the top left corner of the image is drawn.



Note: The **video** element is introduced in the next lesson.

The element can be retrieved from the DOM by using, for example, the **document.getElementById** method. The element can also be created in JavaScript and added to the canvas. However, if the image content is not yet fully loaded, nothing is drawn. This can happen when a page is initializing, or when an image is retrieved dynamically. To ensure the image is drawn correctly onto the canvas, defer the call to the **drawImage** method until after the image is loaded. One way to do this with **img** elements is to handle the **load** event.

The following example shows how to create an **img** element and draw it on the canvas after it is fully loaded.

Creating and Drawing an Image on the Canvas

```
var image = document.createElement("img");
image.onload = function() {
    context.drawImage(image, 20, 20);
};
image.src = "image.png";
```

This code creates an **img** element. Before it assigns the image URL to the element, it assigns an event handler to the **load** event. The event handler draws the image only when it is ready to be drawn. This code produces the following drawing.



FIGURE 5.20: IMAGE

Specifying Width and Height

If the image is retrieved from the DOM, where its width and/or height can be specified, the specified width and/or height are ignored: the image is drawn in its original size. To change the size of the image drawn on the canvas, the second overload can be used.

```
context.drawImage(image, dx, dy, dw, dh)
```

The **dw** and **dh** arguments specify the width and height of the rectangle in which the image is drawn. The image is resized to fit the specified width and height, regardless of its original size. The canvas makes no attempt to preserve the original image's aspect ratio when this overload is used.

The width and height can also be specified as negative values. When a negative width is specified, the image is drawn to the *left* of **dx**. Similarly, when a negative height is specified, the image is drawn *above* **dy**. The image itself is drawn in the same direction; only the coordinates are moved.

The following example draws an image twice, specifying a positive width and height the first time, and a negative width and height the second time.

Drawing an Image with Width and Height

```
var image = document.createElement("img");
image.onload = function() {
    context.drawImage(image, 150, 75, 130, 60);
    context.drawImage(image, 150, 75, -130, -60);
};
image.src = "image.png";
```

The image drawn with a positive width and height has its top-left corner at the center of the canvas, so it appears below and to the right of the center. The image drawn with a negative width and height is drawn in the same direction, with its top-left corner moved up by the width and height, so it appears above and to the left of the center. Because the image is not drawn by using its original dimensions, it appears distorted.



FIGURE 5.21: MULTIPLE IMAGES

Slicing and Scaling an Image

Both of the previously described overloads drew a complete image on the canvas. However, the third and last **drawImage** method overload draws just part of an image. The signature has many arguments.

```
context.drawImage(image, sx, sy, sw, sh, dx, dy, dw, dh)
```

The first four arguments after the **image**, prefixed with an "s" for *source*, specify the source image portion to be drawn. The second group of arguments, prefixed with a "d" for *destination*, designates the area on the canvas where the image is drawn.

This method version can be used to slice and scale images. When an image is sliced, a section is "cut out" and only that section is drawn on the canvas. By multiplying the source image size by some factor and passing that value to the **dw** and **dh** arguments, the image can also be scaled.

The following example draws an image and an enlarged slice.

Zooming in on an Image

```
var image = document.createElement("img");
image.onload = function() {
    var zoomFactor = 2;
    context.drawImage(image, 15, 30);
    context.drawImage(image, 40, 10, 40, 50, 180, 20, 40 * zoomFactor, 50 * zoomFactor);
};
image.src = "image.png";
```

The image on the left is drawn by using the simplest overload. The same image is drawn on the right but zooms in on one of the runners in the picture. Note that the destination's width and height are exactly twice the source slice's width and height: the image on the right is scaled by a factor of 2.

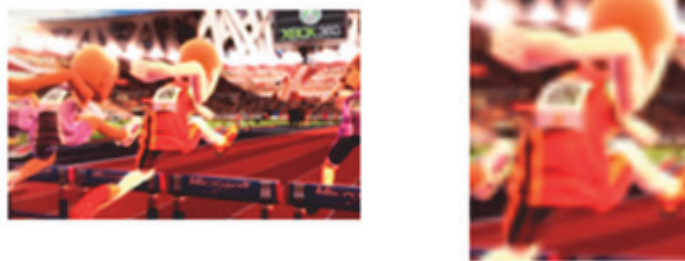


FIGURE 5.22: IMAGE WITH ENLARGED SLICE

The destination arguments, those with the "d" prefix, are identical to the arguments in the second overload. But for anything to be drawn, the whole rectangle described by the source arguments must be inside the source image. In other words, all four rectangle points, from the top-left to the bottom-right, must be valid source image coordinates.

Patterns

Images can be drawn in repeating patterns. Like colors and gradients, patterns are assigned to the **fillStyle** or **strokeStyle** properties and are used to fill or outline arbitrary shapes by using the image. The **createPattern** method creates a pattern that can be assigned to a style. It has the following signature.

```
context.createPattern(image, repetition)
```

The **image** argument specifies any supported image element, similar to the **drawImage** methods. The repetition argument specifies a string similar to the CSS **background-repeat** property. It can have one of the following values:

- "repeat"—specifies that the image is repeated both horizontally and vertically.
- "repeat-x"—specifies that the image is only repeated horizontally.
- "repeat-y"—specifies that the image is only repeated vertically.

- “no-repeat”—specifies that the image appears only once and is not repeated.

The **repeat** argument is not optional, but an empty string can be specified instead of “repeat”, the default value.

A pattern can only be created for an image that has finished loading. If the image is not ready, **createPattern** returns **null**. To ensure the image is ready when creating the pattern, a technique like the one used in the **drawImage** code examples can be used, where the pattern is created in the element’s **onload** event handler.

When a shape is drawn by using a pattern as its style, the pattern begins at the point representing the canvas’ origin, at (0, 0), by default, the top left canvas corner. As the shape is drawn, it “reveals” the part of the pattern where the shape is drawn.

The following example draws a complex shape by using a pattern.

Drawing with a Pattern Fill

```
var image = document.createElement("img");
image.onload = function() {
  var pattern = context.createPattern(image, null);
  context.fillStyle = pattern;
  context.beginPath();
  context.arc(50, 80, 40, 0.5 * pi, 1.5 * pi);
  context.fill();
  context.beginPath();
  context.rect(0, 40, 100, 80);
  context.fill();
  context.beginPath();
  context.arc(150, 80, 40, 0.5 * pi, 1.5 * pi, true);
  context.fill();
};
image.src = "star.jpg";
```

The example is based on a previous example demonstrating linear gradients. In this example, a pattern is used instead of a gradient. The stars begin at the top-left of the canvas, even though the shape itself is drawn far from the corner. This is why some stars on the shape’s edge are partially cut off.



FIGURE 5.23: COMPLEX SHAPE USING A PATTERN

Drawing Text

- The **font** property is set by using a CSS-style declaration
- Text on the canvas uses many CSS paradigms
- Text can be aligned horizontally or vertically
- Multiline text is not supported
- Use the start and end alignments for multilingual support

```
context.fillText(text, x, y);
context.strokeText(text, x, y);
```

Drawing Text

Text is drawn on the canvas similar to how shapes are drawn. The **fillText** and **strokeText** methods are used to draw text. They both have the same signature.

```
context.fillText(text, x, y);
context.strokeText(text, x, y);
```

The **text** argument specifies the string to be drawn. The point specified by the **x** and **y** arguments is an anchor point determining where the text is drawn. The **fillText** and **strokeText** methods draw the text immediately on the canvas and do not affect or interact with the path.

The following example draws some text on the canvas.

Drawing Simple Text

```
canvas.height = 100;
canvas.width = 450;
context.font = "bold 48px Arial";
context.textAlign = "center";
context.textBaseline = "middle";
context.fillText("The canvas is fun!", 225, 25);
context.strokeText("The canvas is fun!", 225, 75);
```

The example prepares the canvas, sets the text's font and style, and then draws the text twice: once with an outline and once without an outline. The **font**, **textAlign**, and **textBaseline** will be covered shortly. This draws the following image.

The canvas is fun!
The canvas is fun!

FIGURE 5.24: SIMPLE TEXT

The canvas cannot draw multiline texts. If the string contains any whitespace characters, including newline characters, they are replaced with simple spaces and the whole string is displayed on a single line. To draw multiline texts, each line is drawn with a separate call to the **fillText** or **strokeText** methods. By default, the text is drawn from left to right, starting at **x**, with the baseline at **y**. However, a few factors affect the text's positioning: the text alignment, the text baseline, and the direction.

Text Alignment

The **textAlign** property specifies the text's horizontal positioning relative to the **x** coordinate passed to the **fillText** and **strokeText** methods. The **textAlign** property has the following signature and possible values:

```
context.textAlign = [ "left", "right", "start", "end", "center" ]
```

When text is aligned to the left, the **x** argument determines where to draw its left edge. When it is aligned to the right, it determines the right edge. When it is aligned to the center, the text is positioned with its center at the **x** coordinate.

The **textAlign** property does not inherit the alignment from the page and cannot be set by using CSS. However, the alignment is affected by the canvas' directionality, which is set by using the CSS **direction** property. The alignment can only be modified by using JavaScript code. The default text alignment is "start".

The "start" and "end" alignments position the text to the left or to the right, depending on the canvas' directionality. The directionality is modified by the CSS **direction** property. If the direction is set to **rtl**, indicating the right-to-left direction, "start" aligns the text to the right and "end" aligns it to the left. The following table shows the actual alignments of each value in different situations.

textAlign Property	Direction	Actual Alignment
left	any	left
right	any	right
center	any	center
start	ltr	left
start	rtl	right
end	ltr	right
end	rtl	left



Note: The canvas can be used for many different purposes requiring text. To accommodate the widest possible audience, use the "start" and "end" horizontal alignments instead of "left" and "right". For websites supporting only one language, this may not be much of an issue. However, as websites grow, they often add multilingual support. Drawing multilingual text on a canvas can be challenging if the possibility is not properly accounted for from the very start, especially for text that goes in the other direction. The "start" and "end" alignments were designed specifically for this purpose, and this is also the reason the default value is "start". The text alignment only solves the problem for text. If the page direction is changed, the text's edge is the opposite edge. It does not apply to other graphics. For example, a chart with labeled

axes and an origin at the bottom-left is still drawn in the same direction. Only the text direction is reversed. This can produce unexpected results and can be a bit tricky to manage sometimes. Careful text alignment management is recommended. Multilingual websites with text going in both directions should always be tested in both directions.

The Baseline

Text is vertically aligned along the baseline, the line on top of which characters rest. Some characters, like "g", descend below the baseline but are still aligned to the baseline. This baseline is known as the *alphabetic baseline*. Text can also be aligned to other baselines, which is very useful when drawing graphics. It is often very convenient, for example, to align multiple strings of text along their middles.

The **textBaseline** property determines the text's vertical alignment relative to the **y** coordinate passed to the **fillText** and **strokeText** methods. The **textBaseline** property has the following signature and possible values, each of which represents an imaginary line.

```
context.textBaseline = [ "top", "hanging", "middle", "alphabetic", "ideographic", "bottom" ]
```

The **textBaseline** property can be set to any of the lines and determines which of these lines is used as the baseline. For example, if "top" is assigned to **textBaseline**, the whole character drops beneath the baseline.

Character sets for different languages use different baselines. For example, character sets for languages using Latin-based characters such as English, are usually aligned to the alphabetic baseline. Languages using ideographs use the ideographic baseline.

- The **top**, **middle**, and **bottom** values correspond to the top, middle, and bottom of the em square, which is the imaginary square in which the character glyph is drawn.
- The **hanging**, **ideographic**, and **alphabetic** values correspond to the common baselines used by most language-specific character sets.

The text baseline is not inherited from the page and cannot be set by using CSS. It can only be set by using JavaScript code. The default baseline is the alphabetic baseline.

The following figure illustrates how different horizontal and vertical alignments are drawn on the canvas. In the image, the horizontal and vertical guides intersecting near each string are the **x** and **y** values passed to the **fillText** method for that string. The vertical guides are drawn over the baselines.

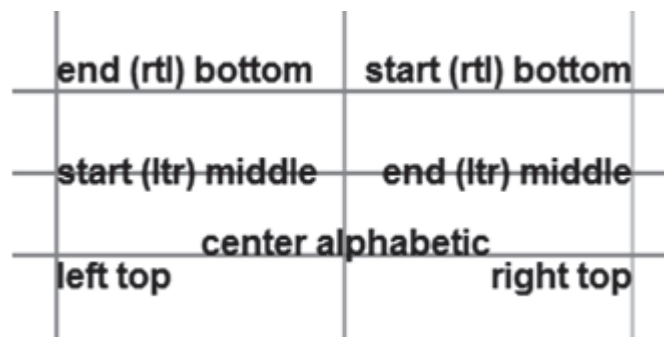


FIGURE 5.25: BASELINES

Note in particular the difference between the first line and the last line. Strings that are aligned to the **bottom** actually appear completely *above* the baseline because their lowermost line is aligned to the

baseline. Similarly, the **top** aligned strings on the last line appear *beneath* the baseline because their upper line is aligned to the baseline.

Fonts and Styles

Text can be drawn on the canvas with almost any font. The **font** property sets the font. It has the following signature.

```
context.font = "style variant weight size family"
```

The passed string syntax assigned to the **font** property is identical to the shorthand CSS **font** property syntax covered in an earlier lesson, although it is interpreted a bit differently.

- The line-height is ignored if specified because the canvas does not support multiline text.
- It does not inherit the CSS font. The font must be specified in code. However, fonts embedded in CSS by using the **@font-face** rule are available to the canvas, after they are finished loading.

The **size** and **family** are mandatory. The **style**, **variant** and **weight** are optional. The **family** must always be specified last, and the **size** must precede the **family**. The **style**, **variant** and **weight** can be specified in any order, as long as they are specified first.



Best Practice: Though the **style**, **variant**, and **weight** can be specified in any order, we recommend that you specify them in the standard order, mainly for consistency. Code having a consistent structure usually has fewer bugs and is easier to maintain.

To set the text color, set the **fillStyle** and **strokeStyle** properties.

The following example uses different fonts on the canvas.

Drawing Fonts

```
context.textAlign = "center";
context.font = "italic small-caps bold 42pt 'Times New Roman', serif";
context.fillStyle = "darkgreen";
context.fillText("Welcome!", 150, 40);
context.font = "900 36px Arial, sans-serif";
context.fillStyle = "blue";
context.fillText("Welcome!", 150, 90);
context.font = "1.5em sans-serif";
context.fillStyle = "yellow";
context.strokeStyle = "red";
context.fillText("Welcome!", 150, 140);
context.strokeText("Welcome!", 150, 140);
```

This code produces the following image.

WELCOME!
Welcome!
Welcome!

FIGURE 5.26: FONTS

Transformations

- Transformations are extremely powerful
- Basic Transformations
 - **scale**
 - **rotate**
 - **translate**
- General Transformations
- Canvas State



```
context.fillStyle = "red";
context.fillRect(10, 10, 30, 20); // draw the red rectangle

context.scale(5, 4);
context.fillStyle = "blue";
context.fillRect(10, 10, 30, 20); // draw the blue rectangle
```

Transformations are an extremely powerful tool for creating impressive graphics. Transformations change the grid, helping to produce many different kinds of drawings and effects.

The canvas exposes a few simple methods for the most common transformations. It also exposes a number of methods allowing programmers to execute a wider variety of more complex transformations.

The Scale Transformation

The **scale** method changes the grid size. The **scale** method determines how many pixels fit in one grid cell. By default, a grid cell is a square that can be drawn with one pixel, but that can be changed. The method has the following signature.

```
context.scale(x, y)
```

The **scale** method's **x** and **y** arguments are not coordinates. They are multiplied by the current grid cell size. The **x** argument is multiplied by the horizontal size, or cell width, and the **y** argument is multiplied by the vertical size, or cell height. If $y = 2$ and $x = 3$, for example, the grid becomes twice as tall and three times as wide.

The following examples draw the same rectangle with two different scales.

Changing the Canvas Scale

```
context.fillStyle = "red";
context.fillRect(10, 10, 30, 20); // draw the red rectangle
context.scale(5, 4);
context.fillStyle = "blue";
context.fillRect(10, 10, 30, 20); // draw the blue rectangle
```

Both rectangles are drawn by using exactly the same arguments. They are both drawn at the coordinates given by (10, 10), have a width of 30, and a height of 20. When the red rectangle is drawn, the grid cell size is 1x1. After the **scale** method is called, the grid cell size is multiplied by 5 for the width, and by 4 for the height.

**FIGURE 5.27: CHANGING SCALE**

To achieve the same result without changing the scale, the blue rectangle is drawn with the statement `context.fillRect(50, 50, 150, 80);`.

Every time the grid is scaled, the **x** and **y** are multiplied by the current horizontal and vertical scales. So if the **scale** method is called again, with the same parameters, the horizontal scale would be 25 ($= 5 * 5$) and the vertical scale would be 16 ($= 4 * 4$). To shrink the scale, simply use numbers between 0 and 1. For example, `context.scale(0.5, 0.5);` shrinks the grid by half.

The Rotate Transformation

Some effects are very difficult to accomplish without rotation but easy when rotation is used. The **rotate** method has the following signature.

```
context.rotate(angle)
```

The **angle** argument is the number of radians to rotate the canvas in the clockwise direction.

Rotating the grid makes it easy to draw shapes at all kinds of strange angles. For example, drawing a square at a 15 degree incline ordinarily requires explicit calculation for each coordinate of each square corner by using the trigonometric sine and cosine functions.

Drawing a square at an incline is very easy when using rotation.

Rotating the Canvas Grid

```
context.rotate(15 * Math.PI / 180); // rotate 15 degrees clockwise
context.fillRect(30, 30, 50, 50); // draw a simple square
```

The only trigonometric calculation required is the simple conversion from degrees to radians.

**FIGURE 5.28: ROTATING THE GRID**

The **rotate** transformation adds the angle to the current rotation. Calling the **rotate** method with the same arguments as in the example, places the grid in a final rotation of 30 degrees from 0. To rotate in a counterclockwise direction, use a negative angle.

The Translate Transformation

Translating is simply “moving”. The **translate** method *moves* the origin, the (0, 0) coordinates to a different location. The **translate** method has the following signature.

```
context.translate(x, y)
```

The **x** and **y** arguments are not coordinates. They represent the number of grid cells to move the origin horizontally and vertically. By default, the origin is at the canvas's upper-left corner. However, it can be moved anywhere.

The following example moves the origin by using the translate method.

Using the Translate Transformation

```
context.strokeRect(0, 0, canvas.width, canvas.height); // draw a frame around the canvas
context.translate(canvas.width / 2, canvas.height / 2); // move the origin to the center of the canvas
context.arc(0, 0, 50, 0, 2 * Math.PI); // draw a circle at the center
context.fill();
```

This method demonstrates one of the more common uses of the **translate** transformation. Many calculations when working with graphics are simpler when they do not have to be adjusted for location. Moving the origin to where the calculations are the easiest, makes the code easier to write, read, and understand.

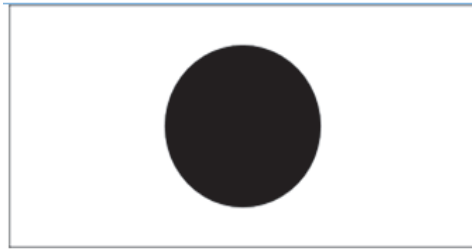


FIGURE 5.29: TRANSLATE TRANSFORMATION

Very often, scale, translation, and rotation are used together.

Using Different Transformations Together

```
context.strokeRect(0, 0, canvas.width, canvas.height); // draw a frame around everything
context.translate(canvas.width / 2, canvas.height / 2);
context.scale(4, 2); // enlarge the grid and stretch it horizontally
context.arc(0, 0, 30, 0, 2 * Math.PI); // draw a circle skewed by the scale method to look like an ellipse
context.fill();
context.rotate(Math.PI); // turn upside down
context.fillStyle = "orange";
context.textAlign = "center";
context.textBaseline = "middle";
context.fillText("upside down", 0, 0); // draw the text "upside down" upside down
```

This example uses a number of simple transformations to create a complex shape. The translate method moves the origin from the top left corner to the center of the canvas. The scale method then enlarges the grid by multiplying the horizontal axis by 4 and the vertical axis by 2. This not only enlarges but also stretches the horizontal axis twice as much as the vertical axis. The arc method then draws a circle with a radius of 30. Because the grid is scaled and stretched, the circle is also scaled and stretched to look like an ellipse. Finally, the grid is rotated 180 degrees so the "upside down" text actually appears upside down. This example produces the following image.

**FIGURE 5.30: MULTIPLE TRANSFORMATIONS**

General Transformations

Transformations are actually calculated internally by using matrices. When multiple matrices are used, they are multiplied by each other. Matrices are beyond the scope of this lesson, but the important thing is that calculating a matrix requires multiple mathematical operations and all the applicable matrices have to be calculated for each pixel on the canvas. Using many transformations in graphically intensive canvas apps can significantly impact performance. Browsers may help improve the performance by sending the matrices to the GPU, which is optimized for calculating matrices and specifically transformations. However, not all browsers are the same and they may not always use hardware assistance.

Another effective and more predictable technique to improve performance is to combine transformations. Each transformation, including the **scale**, **rotate** and **translate** transformations, has its own matrix. By combining matrices, fewer intensive calculations are required.

The **transform** method is a general-purpose transformation method. The **transform** method has the following signature.

```
context.transform(a, b, c, d, tx, ty)
```

The arguments represent the values passed to the transformation matrix.

In addition to combining matrices, the general transformation method can be used to create more complex transformations that can skew the canvas in different ways.

Just as the **scale**, **rotate**, and **translate** methods modify the current matrix, so does the **transform** method. To "start over" with a clean matrix, the **setTransform** method can be used. It has a similar signature to the **transform** method and does the same thing, except that it first initializes the current transformation matrix to the *identity* matrix, which is the initial canvas state.

```
context.setTransform(a, b, c, d, tx, ty)
```

Canvas State

When applying many transformations, it can be easy to lose track of which transformations are currently applied. The canvas provides a simple mechanism to remember the state of most of the canvas' settings, including the transformations. The state mechanism works on a stack and uses two methods performing the equivalent of the *push* and *pop* operations.

```
canvas.save()  
canvas.restore()
```

Both methods have no parameters and return no values. The **save** method pushes all the current settings to the top of the stack for later retrieval and the **restore** method pops them. After they are popped from the stack, they automatically replace all the current settings.

The following example uses the state stack to more conveniently handle settings and transformations.

Using the State Stack

```
context.font = "bold 14pt Arial";
context.textAlign = "center";
context.textBaseline = "middle";
context.save(); // push the text settings to the state stack
context.translate(10, 10);
var length = 66;
for(var i = 0; i < 4; i++) {
    context.save(); // push the current position to the state stack
    context.fillRect(0, 0, length, length);
    context.save(); // push the current fill settings to the state stack
    context.translate(length / 2, length / 2);
    context.fillStyle = "orange";
    context.scale(3, 2);
    context.fillText(i + 1, 0, 0);
    context.restore(); // restore the previous fill style, location and scale from the
state stack
    context.translate(2, length - 5);
    context.fillStyle = "yellow";
    context.fillRect(0, 0, length - 4, 3);
    context.restore(); // restore the previous fill style and location from the state stack
    context.translate(length + 4, 0);
}
context.restore(); // restore the original settings, including the base font and
position, from the state stack
context.translate(canvas.width / 2, length + 30);
context.fillText("pick a number", 0, 0)
```

Every successive group of calls in the example is “wrapped” between a call to the **save** method and a call to the **restore** method. This helps provide some isolation. Changes made to the font, the fill style, or the transformation matrix, only apply as long as the settings are current. For example, each number is skewed by using the **scale** method and written by using a different text color. Also, the actual pixel position of the text never has to be calculated because the successive translations place the origin at exactly the right spot to draw the text. This code produces the following drawing.



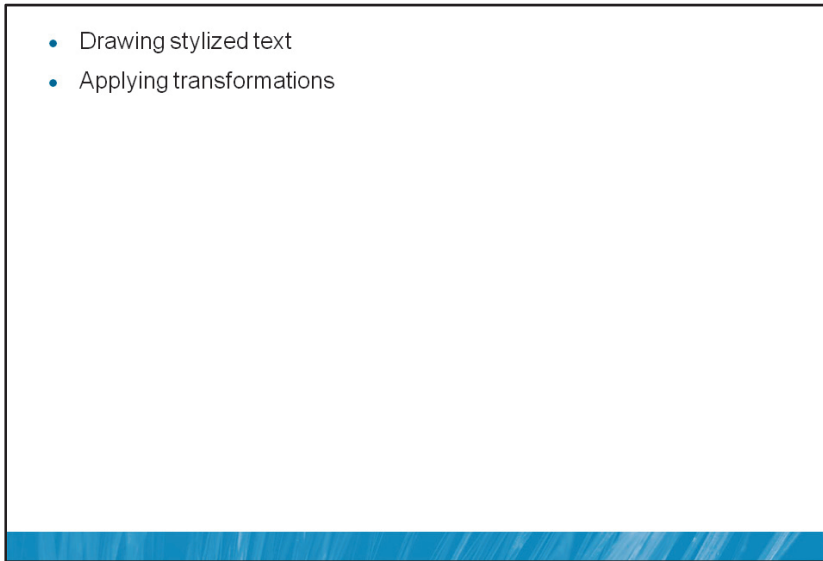
FIGURE 5.31: STATE STACK



Best Practice: In a real application, each wrapped save/restore section might be contained in its own function, with the **save** and **restore** calls. It is highly recommended to make use of the **save** and **restore** methods, providing a very simple and effective mechanism for making quick isolated localized changes.

The state stack is especially useful when managing multiple successive transformations, but can be used with almost any setting. The stack stores many 2d context's settings, including the transformation matrix, fill and stroke styles, clipping regions, line settings, and font and text settings.

Demonstration: Drawing and Transforming Text



This demonstration shows how to:

- Draw text on a canvas.
- Transform the canvas grid.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file, add a **canvas** element inside the **body** element.

```
<canvas id="canvas">
</canvas>
```

4. In the HTML file, add a **script** element with a **draw** function inside the **head** element, before the **head** element's closing tag.

```
<head>
...
<script type="text/javascript">
  function draw() {
    var canvas = document.getElementById("canvas");
    var context = canvas.getContext("2d");
  }
  document.addEventListener("DOMContentLoaded", draw, false);
</script>
</head>
```

5. At the end of the **draw** function, draw a circle skewed to look like an ellipse and translated to the center of the canvas.

```
<script type="text/javascript">
  function draw() {
    var canvas = document.getElementById("canvas");
```

```

var context = canvas.getContext("2d");
context.translate(canvas.width / 2, canvas.height / 2);
context.scale(4, 2); // enlarge the grid and stretch it horizontally
context.arc(0, 0, 30, 0, 2 * Math.PI); // draw a circle skewed by the scale
method to look like an ellipse
context.fillStyle = "orange";
context.fill();
}
document.addEventListener("DOMContentLoaded", draw, false);
</script>

```

6. At the end of the **draw** function, draw text and rotate it.

```

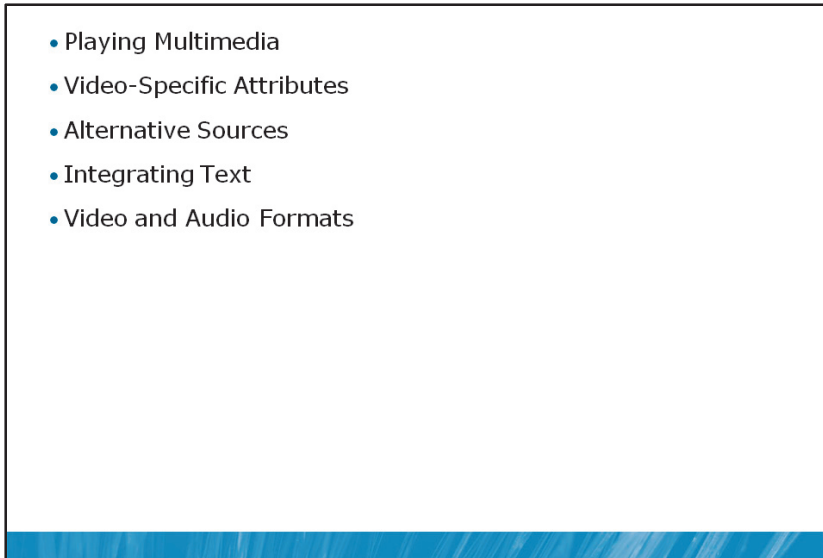
<script type="text/javascript">
function draw() {
    var canvas = document.getElementById("canvas");
    var context = canvas.getContext("2d");
    ...
    context.rotate(Math.PI); // turn upside down
    context.font = "bold 9px Arial";
    context.fillStyle = "blue";
    context.textAlign = "center";
    context.textBaseline = "middle";
    context.fillText("upside down", 0, 0); // draw the text "upside down" upside down
}
document.addEventListener("DOMContentLoaded", draw, false);
</script>

```

7. Save the HTML file.
8. Run the saved file.

Lesson 2

Video/Audio Formats and Codecs



Just as add-ons were used for graphics, so were they used for multimedia. HTML5, however, provides innate support for multimedia playback on webpages by using the **video** and **audio** elements, instead of add-ons. These elements have additional advantages over the traditionally used add-ons:

- They have many accessibility features that can benefit people, for example, with low vision or hearing.
- They can be synchronized with each other very easily, without using code.
- They are fully scriptable by using JavaScript, so programmers do not need to resort to proprietary languages and tools to control playback.
- They support multiple media tracks that support user-selectable multilingual multimedia and also "commentary"-style tracks similar to those commonly found on DVDs.
- They are part of HTML5, so any HTML5 browser can play them without requiring a possibly unavailable add-on.

Different browsers and operating systems support different multimedia formats and codecs. The HTML5 standard accounts for these differences so that the widest audience can experience the multimedia. Web developers can make available versions of their multimedia files in multiple formats so that each browser can automatically select and play the version it supports.

This lesson shows how to use the **video** and **audio** elements, and provides an introduction to working with multimedia formats and codecs.

Lesson Objectives

After completing this lesson, you will be able to:

- Play video and audio by using HTML.
- Use video-specific attributes.
- Use alternative media sources.
- Integrate text for subtitles, accessibility, and other purposes.
- Choose which video and audio formats to use.

Playing Multimedia

- The **video** and **audio** elements are interchangeable
- The media file is specified with the **src** attribute
- The **autoplay** and **loop** attributes control automatic playback
- The **preload** attribute recommends a buffering strategy
- The **volume** and **muted** attributes control the volume
- The **controls** attribute toggles the built-in media controls
- Add fallback content to support legacy browsers

```
<video src="video.m4v"></video>
<audio src="audio.m4a"></audio>
```



Using Media Elements

HTML5 introduces two media elements—the **video** and **audio** elements—which let browsers play video and audio content. The two elements have some obvious differences. For example, the **video** element displays a video's pictures and the **audio** element displays nothing. However, they also have much in common:

- *Both* elements play audio data. Videos usually have one or more audio tracks that are played with the pictures.
- Both elements contain a lot of data, since video and audio files can be very large.
- Users interact with both video and audio in much the same way. For example, users may change the volume, or pause and resume playback.

Furthermore, as far as the browser is concerned, whether a media file contains video or audio is immaterial. What matters is whether the page uses a **video** or **audio** element to play the media. If a video is played by using an **audio** element, for example, the browser only plays the video's audio content.

Both elements have many common attributes. The following sections describe the common attributes. Video is more complex, so the video-specific attributes are covered in a later topic.



Best Practice: Although the **video** and **audio** elements can technically be used interchangeably with video and audio resources, we do not recommend using them in this way. In some cases, as in the example of using a video file with an **audio** element, the browser may not be able to download only the tracks needed to play just the audio. This can lead to excessive downloads and slow playback and can detrimentally affect the user experience. Unless there is a special reason to use them interchangeably, the **video** element should only be used for videos and the **audio** element should only be used for audio.

Specifying the Media Source

In their simplest form, only the **video** or **audio** tags and the **src** attribute are needed to add multimedia to the page.

The following examples specify the multimedia files to use.

Using the src Attribute

```
<video src="video.m4v"></video>
<audio src="audio.m4a"></audio>
```

The **src** attribute is similar to the **src** attribute of the **img** element and specifies the URL of the file to play.

Automatic Playback

The browser does not automatically start playing media unless the **autoplay** attribute is used. The **autoplay** attribute is a Boolean attribute that determines whether the video or audio should be played as soon as the page is loaded. If present, the browser automatically starts playback, and if absent, automatic playback is disabled. By default, automatic playback is disabled.

The following examples instruct the browser to play the video and audio automatically.

Using the autoplay Attribute

```
<video src="video.m4v" autoplay="autoplay"></video>
<audio src="audio.m4a" autoplay="autoplay"></audio>
```

The **loop** attribute can be used to automatically replay the media from the beginning. It is also a Boolean attribute. When absent and the video or audio reaches the end, it stops. When present and playback reaches the end, the browser automatically starts playing the video or audio again from the beginning. By default, looping is disabled.

The following attribute is used to automatically loop video and audio playback.

Using the loop Attribute

```
<video src="video.m4v" loop="loop"></video>
<audio src="audio.m4a" loop="loop"></audio>
```



Note: The loop attribute only applies while the video or audio is playing. It does not automatically start playing when the page or the media is loaded; it only goes back to the beginning after reaching the end. To start playing automatically and have the media repeat indefinitely, use both the **autoplay** and **loop** attributes together.

Buffering

Video and audio files generally contain a lot of data that has to be downloaded while playing. To play them well, without pausing or having to wait for the browser to finish downloading, browsers have to buffer the data. Buffering entails downloading some of the data before playing the video or audio content. Buffering significantly improves the user experience.

However, it may not always be a good idea to use buffering. For example, multimedia can be played over networks with limited bandwidth or where large downloads can be prohibitively expensive. Buffering might also require a lot of CPU resources and storage space for the buffered data, which can significantly slow down devices with limited resources.

Web developers can use the **preload** attribute to specify the kind of buffering the browser should use. The **preload** attribute describes the web developer's *intentions*. The browser may ignore the suggestion if it runs in an environment where the suggestion does not make sense, if it can detrimentally affect the user experience, or if user settings override it.

The preload attribute has three possible values:

- "none"—The browser should not buffer the data at all. This can be useful, for example, on pages that have many videos, or for pages that are intended to be used a lot by smartphones.
- "metadata"—The browser should download only the data portion that contains the metadata, usually found at the beginning of the file. This can be useful, for example, on constrained networks.
- "auto"—The browser should download as much of the data as possible to ensure the user has the best possible experience, even if it takes up a lot of system resources.

When the preload attribute is specified and its value is empty (e.g., `preload=""`), it is equivalent to specifying the value "auto". When the attribute is not specified at all, each browser defines its own default value.

The following example advises the browser that it should only download the metadata.

Using the preload Attribute

```
<video src="video.m4v" preload="metadata"></video>
<audio src="audio.m4a" preload="metadata"></audio>
```

Playback Controls

To let users control video and audio playback, browsers can show playback controls. Playback controls may include, for example, play, pause and stop buttons, a volume slider and a seek bar. Every browser can implement its own playback controls. However, Web developers may choose to implement their own controls, whether to implement controls with application-specific logic, or to ensure the controls share the same look and feel as the rest of the site, or simply to allow every user to use the same controls on every browser.

The **controls** attribute is used to specify whether the browser displays the built-in controls. The **controls** attribute is a Boolean attribute: the built-in controls are only shown if the attribute is present. By default, controls are not visible.

The following example instructs the browser to display the built-in controls.

Using the controls Attribute

```
<video src="video.m4v" controls="controls"></video>
<audio src="audio.m4a" controls="controls"></audio>
```

Though the **audio** element has no visual representation, browsers may show the audio controls outside of the page, for example, on a menu. In addition, browsers may choose to ignore the **controls** attribute in some cases or on some platforms.



Note: When web developers wish to implement their own controls, they may toggle the built-in controls off by leaving the **controls** attribute unspecified. Implementing the controls requires JavaScript, which is covered in the next lesson.

Volume

Two attributes control audio volume for both the **video** and **audio** elements: **volume** and **muted**.

- The **volume** attribute specifies how loud the audio should be. It can be any value between 0 and 1 inclusive, where 0 is silent and 1 is the loudest volume. The value must not be outside this range.
- The **muted** attribute is a Boolean attribute that determines whether the audio is muted. When absent, the volume is determined by the **volume** attribute, but when present, it overrides the **volume** attribute and silences the playback.

The following example sets the volume to 50 percent, but mutes it.

Using the volume and muted Attributes

```
<video src="video.m4v" controls="controls" volume="0.5" muted="muted"></video>
<audio src="audio.m4a" controls="controls" volume="0.5" muted="muted"></audio>
```

In the example, the attributes set the volume at 50 percent of the loudest volume and mute the audio. If the user unmutes them, by using the built-in controls, their volume will return to 50 percent.



Note: By default, audio is unmuted and the volume is 1, which is the loudest volume. However, browsers very often remember user settings across different pages and sites and those settings override the defaults. For example, if a user mutes the audio for some video, and then watches another video, the browser may mute the second video as well. In other words, the default volume is unpredictable.

Despite the unpredictable default value, most applications should not explicitly set the volume. Users are accustomed to controlling volume and the Web developer can never know in which environment the video or audio is played. Explicitly setting the volume can override user settings on some browsers and may annoy many users.



Best Practice: Explicitly setting the volume in HTML markup should be reserved only for those cases where not doing so may damage the user experience. For example, pages that use the **video** element for advertisements or show many videos on a page should probably mute them until the user manually plays them.

Fallback Content

Earlier browsers may not support the **video** and **audio** elements and users may disable them on browsers that do. To show users content by using legacy mechanisms, such as add-ons, or to show instructions on how to play the video and audio, web developers can add fallback content inside the **video** and **audio** elements, as child elements. Browsers that recognize the elements will not show the fallback content. Browsers that do not recognize them will display only the fallback content. This is an effective mechanism to support users across a wide range of platforms and devices.



Best Practice: The **source** and **track** elements, covered a little later, are also added as children of the **video** and **audio** elements. They should always be added first, before any fallback content.

The following example instructs the user to download an updated browser on browsers that do not support the video and audio elements.

Adding Fallback Content

```
<video src="video.m4v">
  Your browser is unable to play this video content. Please download a more recent
  browser to see the video.
</video>
<audio src="audio.m4a">
  Your browser is unable to play this audio content. Please download a more recent
  browser to hear the audio.
</audio>
```



Note: The **audio** element has no visible representation on browsers that recognize the element. Browsers that do not recognize it and do not know that they should to hide it, will display the fallback content

Video-Specific Attributes

- The **video** element can be laid out easily by using CSS
- The **width** and **height** attributes specify the dimensions of the viewing area
- The video always maintains its innate aspect ratio
- The **poster** frame image is shown before the video is played

```
<video src="video.m4v" poster="first_frame.png"
width="100px" height="100px"></video>
```



Layout

Unlike the **audio** element, the **video** element has a visual representation: it displays a viewing rectangle that contains the playing video. Unlike content that is embedded by using the **object** element and relies on add-ons, the **video** element flows more naturally with the page content and can be laid out very easily with CSS. The **video** element also has a number of unique attributes that define its behavior.

Width and Height

When the video element is placed on a webpage, the browser checks the width and height of the video itself and plays the video in a viewing rectangle of the same size. If no video is specified, the video element's default size is 300 pixels by 150 pixels.

To control the size of the video on the page so it fits better with the rest of the page content, the **width** and **height** attributes can be used. The **width** and **height** attributes specify the size of the viewing rectangle that contains the video in pixels.



Note: There is a significant difference between the *element's* width and height, and the *video's* width and height. The width and height of the element are the size of the container that shows the video on the page. However, the video itself may be created with different dimensions. Keeping the distinction in mind is very important.

If only one of the attributes is specified, the browser automatically sizes the other one to match the aspect ratio of the video itself. The aspect ratio is simply the video's width divided by its height. For example, if the video is 400 pixels wide and 200 pixels high, its aspect ratio is 2 (because 400 pixels / 200 pixels = 2). If the Web developer sets the **width** attribute to 300 pixels, the browser automatically sets the height of the element to 150 pixels (because 300 pixels / 2 = 150 pixels), to match the video's aspect ratio. If only the **height** attribute is set, the width is automatically sized in the same way. The browser may shrink or expand the viewing rectangle to ensure that the aspect ratios match.

Specifying just one of the size attributes ensures that the video is never distorted and that it fits perfectly inside the viewing rectangle.

Maintaining the Aspect Ratio of the Video

```
<video src="video.m4v" width="400"></video>
```

If both the **width** and **height** attributes are specified, the browser cannot resize the viewing rectangle. Instead, it will resize the video so it fits inside the viewing rectangle, without changing the video's aspect ratio. For example, if the video element that shows the 400 x 200 pixel video in the previous code example has its **width** attribute set to 300 pixels and its **height** attribute set to 100 pixels, the aspect ratios do not match because the video has an aspect ratio of 2 and the viewing rectangle has an aspect ratio of 3. The browser cannot resize the viewing rectangle, so it will resize the video itself. The video's height will be 100 pixels, its width will be 200 pixels (because 100 pixels * 2 = 200 pixels) and the video will be centered inside the viewing rectangle, leaving 50 empty pixels on each side of the video.



Best Practice: Sometimes the **width** and **height** attributes must be used for technical reasons, but usually the size of the video element on the page is a layout concern, so the width and height should be specified by using CSS. In fact, if both the attributes and the CSS properties are specified, the CSS properties override the attributes.

The Poster Frame

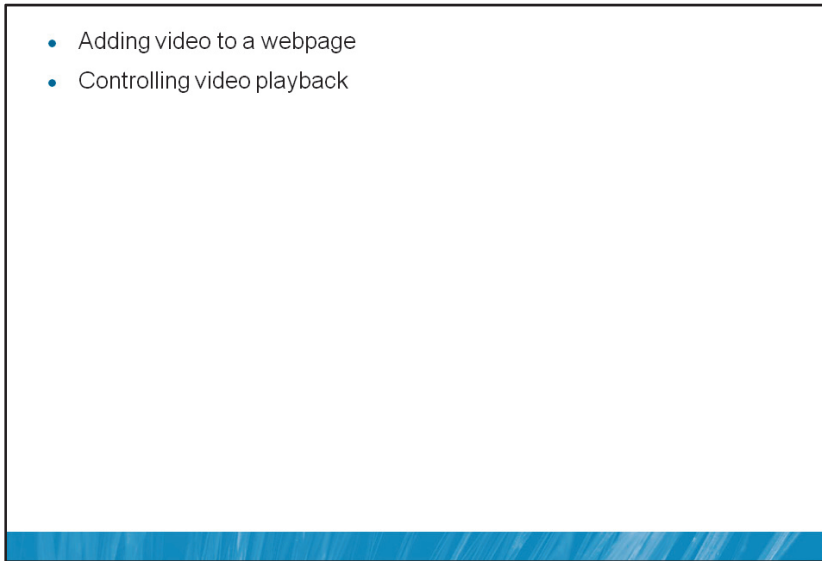
While the video is loading for the first time, or before it is played (manually by the user or automatically by the browser), the browser can show a poster frame. The poster frame is an image that can give the user an idea of what the video contains. The poster frame is specified by using the **poster** attribute and contains the URL of an image. By default, the browser does not display a poster frame.

The poster attribute is often used to show an image of the video's first frame.

Using the poster Attribute

```
<video src="video.m4v" poster="first_frame.png"></video>
```

Demonstration: Playing Video



This demonstration shows how to:

- Add video to a webpage.
- Lay the **video** element out on the webpage.
- Display the built-in video controls.
- Automatically loop the video.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Add the file, **D:\Mod05\Democode\Assets\trailer.m4v**, to the web application by copying or dragging the file to the new project's folder in Solution Explorer.
3. Create an HTML file.
4. In the HTML file, add a **video** element with built-in controls supporting automatic playback and looping inside the **body** element.

```
<video src="trailer.m4v" width="400" autoplay="autoplay" loop="loop"
controls="controls">
</video>
```

5. Save the HTML file.
6. Run the saved file.

Alternative Sources

- The **source** element specifies alternate media resources
- Multiple **source** elements can be specified
- The browser uses only the first **source** element it supports
- Specifying the MIME type with the **type** attribute avoids downloading unsupported files
- The **media** attribute constrains the environment by using *media queries*

```
video>
  <source src="video.m4v" " type="video/mp4" />
</video>
<audio>
  <source src="audio.m4a" " type="audio/mp4" />
</audio>
```

Specifying Alternative Media Sources

When the **src** attribute is specified, the **video** and **audio** elements consume whatever media the attribute refers to. However, different browsers support different types of media. If the **src** attribute is used, browsers that do not recognize the type of media it refers to will not be able to play it.

Multiple **source** elements can be used instead of the parent element's **src** attribute to specify different versions of the video or audio resource. To use the **source** element, the **src** attribute must be omitted from the parent **video** or **audio** element because it always overrides the **source** elements. Any number of **source** elements can be added as immediate children of the **video** or **audio** element. Each **source** element specifies a different version of the media resource.

The following examples specify multiple video and audio sources.

Using the source Element and its src Attribute

```
<video>
  <source src="video.m4v" />
  <source src="video.webm" />
  <source src="video.ogv" />
</video>
<audio>
  <source src="audio.m4a" />
  <source src="audio.ogg" />
  <source src="audio.wav" />
</audio>
```

The source Element's src Attribute

As shown in the previous example, the **source** element's **src** attribute specifies the URL of the version of the resource the **source** element represents. The order in which the sources are specified is important. When a browser parses the **source** elements, it checks them in the order they appear in the document, one after another, until it finds one that it supports. In general, it downloads only the media resource it actually supports and discards the others.

However, if only the **source** element's **src** attribute is specified, the browser lacks information to determine the media's type with certainty. Browsers might try to guess based on the file extension or they may download the file or just part of it (the metadata) until they have enough information to determine whether they support the file or not. This process is unpredictable and may change from browser to browser. A better option is to explicitly specify the media's type.

The type Attribute

The **type** attribute is used to explicitly specify the media type the **source** element represents. Instead of downloading each file in turn, possibly wasting valuable time and bandwidth, the browser can examine the **type** attribute, which is a string that specifies the media file's MIME type. If the browser does not recognize the MIME type, it can simply skip over the **source** element without downloading anything and try the next one.

The following example uses the type attribute to specify an MP4 video file and an Ogg audio file.

Using the type Attribute

```
source src="video.m4v" type="video/mp4" />
<source src="audio.ogg" type="audio/ogg" />
```



Note: Though the **type** attribute specifies the media resource's MIME type as either having a video or audio format, both **source** elements in the example can be used with either **video** or **audio** elements because they are interchangeable.

The media Attribute

The **media** attribute also helps the browser determine whether the file the source element refers to is appropriate and should be downloaded. The media attribute is a string that contains a *media query*. A media query is an expression that specifies the environmental conditions in which the video or audio file can be played.

The following example uses the media attribute to select one video resource designed for small screens and another that can be played on larger screens .

Using the media Attribute

```
<video>
  <source src="video.small.m4v" type="video/mp4" media="max-device-width: 400px" />
  <source src="video.m4v" type="video/mp4" />
</video>
```

The example declares a **video** element with two **source** elements that have exactly the same MIME type. When the browser checks the **source** elements, it checks them in the order they appear. The first **source** element has a media query that specifies its media resource can only be played on a device 400 pixels wide or less. If the device is 400 pixels wide or less, it matches the media query and the browser downloads the "video.small.m4v" file, which is presumably optimized for that kind of device. For every other device, the media query will return *false*, so the browser tries the next **source** element even if it supports the MIME type.

If the media attribute is unspecified or empty, its value is equivalent to "all", which determines the source element can be used on *all* platforms. The default value is "all".

Media queries can also be used to determine if the user's browser implements certain accessibility features, such as supporting a Braille reader or a screen reader. This can be used with other accessibility features to download special media resources designed for use with those browsers.

Integrating Text

- The track **element** associates timed text with a video
- The standard text track file formats are WebVTT and TTML
- The kind attribute specifies the role of the text track
- The srclang attribute specifies the text track's language
- Subtitle, caption, and description tracks are listed by using their **label** attribute
- The track marked with the **default** attribute is automatically selected on load

```
<video src="video.m4v">
  <track src="video.vtt" />
</video>
```

The track Element

Multimedia files generally contain a lot of important information represented as binary data. This makes them relatively difficult to use on the web. Search engines have great difficulty indexing them. They cannot be used by many people with low vision or hearing or other disabilities. In a global context, the audio may not be understood by people who speak other languages. In addition to these problems, additions that are used to play video and audio provide no standard, or even popular mechanism to deal with these issues.

HTML5 introduces the **track** element to solve these problems. The **track** element is used to associate text with multimedia resources. The text may contain subtitles, closed captions for people with low hearing, translations, transliterations, or other relevant textual content. It can be easily indexed by search engines, is designed for accessibility, and provides a single standard mechanism any web developer can use. Multiple track elements can be used to represent different "text tracks", each serving a different purpose. The **track** element is added as a child of the **video** and **audio** elements, after the **source** elements and before the fallback content.

The track Element's src Attribute

Tracks represent text files that contain associated text. The **src** attribute contains a URL that specifies where to download the text file.

To add an associated text file, use the track element and specify its source with the src attribute, as the following example shows.

Using the track Element with its src Attribute

```
<video src="video.m4v">
  <track src="video.vtt" />
</video>
```

Text Track File Formats

Web developers can associate any text file with a **video** element by using a **track** element. There is no guarantee that all browsers recognize all the file formats, though special add-ons may be used to support different file formats. The two text file formats that are part of the HTML specification as of this writing and most likely to be supported by the greatest number of modern browsers, are the *Web Video Text Track (WebVTT)* and the *Timed Text Markup Language (TTML)* formats.

The text track in the previous example uses the .vtt file extension, which is the file extension for WebVTT files. WebVTT files are somewhat similar to text files using other subtitle formats, such as SRT files, but WebVTT is part of the HTML5 standard and designed specifically to mark up text files for use with the **video** element. For every line of text, the WebVTT file specifies the precise time in the video associated with that text and provides some instructions (cues) the browser can use to display the text, such as the position of the text overlaid on the video.

The TTML format is an XML format with the similar goal of standardizing text/video synchronization. Whereas WebVTT files are plain text files that are easy for people to read and compose, TTML files are XML files that are well-suited for use in code.



Note: There are many text file formats for subtitles and captioned text. However, most of them are not standardized and are not designed for the breadth of problems WebVTT and TTML file formats aim to solve. There is also no guarantee that browsers will support nonstandard formats. It is strongly recommended to use one of the standardized timed text file formats.

The *kind* and *srclang* Attributes

By default, text track files are used for subtitles. However, text tracks can also be used for other content. The *kind* attribute specifies the text file's role. It can have one of the five following values:

- “subtitles”—Subtitles are used mainly for translating speech into other languages. Browsers generally overlay subtitles on the video itself.
- “captions”—Captions are similar to subtitles but are intended for environments in which sound cannot be used or for people with low hearing. The “hoh” designation is often used for caption files to designate they are for the “hard-of-hearing”. They too are usually overlaid on the video.
- “descriptions”—Subtitles and captions provide information about the audio component of the video. Descriptions, on the other hand, provide information on the visual component. They can be used to describe, for example, the scene at any given moment. They are designed to be synthesized as audio so computers can automatically read them to listening users and are intended for situations in which a screen is unavailable or cannot be used, and for people with low vision.
- “chapters”—Chapters are used for navigating the video. The browser can show the chapter titles to make it easy for users to jump from one chapter to another.
- “metadata”—Metadata tracks are designed to be used by JavaScript code. Web developers can generate, for example, metadata tracks they can later use to synchronize code with events displayed in a video, such as synchronizing a slideshow with an on-video speaker.

The default value is “subtitles”.

When the text track contains subtitles, the **srclang** attribute must be used to specify the text language, although it can also be used for other kinds of text tracks. The **srclang** attribute does not have a default value.

The following example adds a few subtitle tracks and a metadata track to synchronize a slideshow.

Using the kind Attribute

```
<video src="video.m4v">
  <track src="video.en.vtt" kind="subtitles" srclang="en" />
  <track src="video.en.hoh.vtt" kind="captions" srclang="en" />
  <track src="video.fr.vtt" kind="subtitles" srclang="fr" />
  <track src="video.slides.vtt" kind="metadata" />
</video>
```

The label Attribute

The label attribute is used to assign titles to text tracks. The titles may be displayed by the browser when it lets users choose which text tracks to use. The label is only applicable to “subtitles”, “captions”, and “descriptions” text tracks, which may be shown by the browser.

The following example adds labels to some of the text tracks.

Using the label Attribute

```
<video src="video.m4v">
  <track src="video.en.vtt" kind="subtitles" label="English subtitles" />
  <track src="video.en.hoh.vtt" kind="captions" label="English subtitles - closed
captioned" />
</track>
```

The label attribute has no default value, and if left unspecified, different browsers may show different labels.

The label attribute must be unique for a specific kind and language. In other words, if two tracks have the same values for the **srclang** and **kind** attributes, the **label** attribute must be different. However, if two tracks with different languages are of the same kind, their labels can be similar.

The default Attribute

The **default** attribute is used to specify which track the browser should use by default, if any. The **default** attribute is a Boolean attribute. Only one track may use the **default** attribute.

The following example specifies that English subtitles are to be used by default.

Using the default Attribute

```
<video src="video.m4v">
  <track src="video.en.vtt" kind="subtitles" default="default" />
  <track src="video.fr.vtt" kind="subtitles" />
</track>
```

Video and Audio Formats

- A codec encodes and decodes video or audio
- The MIME type string can specify both the container *and* the codec
- Use multiple **source** elements to support the most platforms
- The Dominant Video Codecs:
 - MP4/H.264: highest quality, narrowest support
 - WebM/VP8: lower quality, wider support
- The Dominant Audio Codecs:
 - MP3: lower quality, widest support
 - AAC: high quality, wide support
- Vorbis: high quality, wide support



```
<video>
  <source src="video.ogv"
    type='video/ogg; codecs="theora,
    vorbis"' />
</video>
```

Video Codecs and Containers

A video codec is a software library that can encode and decode video by using a specific algorithm. The different algorithms are optimized for different purposes. For example, some codecs may optimize the video for transfer and playback over low bandwidth connections, while others may optimize it for greater fidelity.

Video codecs encode video data and place that data inside a container that is used to distribute the video. The container is like a compressed archive file that contains the encoded video data along with metadata. Some containers can hold all kinds of data, including multiple video, audio and text tracks encoded by using various codecs, whereas others are specifically designed to contain a limited number of tracks or only tracks encoded by using a specific codec. While most containers are general-purpose containers that can contain video encoded by using various codecs, usually, specific codecs are used with specific containers.

The codecs MIME Type Parameter

The **type** attribute of the **source** element uses a MIME type string to specify the video format. The MIME type strings in the previous examples all specified the type of container and not the type of codecs. Because some containers can contain video encoded with almost any codec, specifying only the type of container may not provide the browser with enough information to determine which specific codec was used to encode the video. To prevent guesswork, the exact codecs can be specified by adding the **codecs** parameter to the MIME type. To add the codecs parameter, a semicolon is first appended to the MIME type string, followed by the codecs parameter and a list of comma-separated codecs. Multiple codecs can be used in a container holding multiple tracks, such as separate video and audio tracks. The comma-separated list details the codec used for each track.

The following example specifies the precise container and codec used by the video file referenced by the source element.

Using the MIME Type codecs Parameter with the source Element

```
video>  
  <source src="video.ogv" type='video/ogg; codecs="theora, vorbis"' />  
</video>
```

The complete MIME type string in the example is 'video/ogg; codecs="theora, vorbis"'. It is written inside single quotation marks to avoid conflicts with the double quotation marks used by the **codecs** parameter. The first part, "video/ogg" specifies that this **source** element refers to a video stored in an Ogg container. The second part specifies the list of codecs used to encode and decode the video. In this case, the Theora video codec is used to encode the video track and the Vorbis audio codec is used to encode the audio track.

The Dominant Video Codecs

Different browsers and operating systems support different video codecs. For various reasons, such as licensing restrictions, technical limitations, and the large number of different devices and platforms, there is no consensus on which video codec should be used as the common standard. But while there may not be a single universal video codec, there are a small number of very popular video codecs.

This list is not exhaustive and describes only the most popular video codecs in use.

The MP4/H.264 Codec

The H.264 codec is a proprietary codec that produces very high quality video with very small files. Video encoded with this codec is usually stored in an MP4 container. The H.264 codec actually refers to a family of related codecs optimized for different kinds of tasks. For example, one of the codecs is optimized for streaming video and another is optimized for storage.

The H.264 codec is supported by Internet Explorer and Safari, as well as all of the Microsoft and Apple HTML5-compliant smartphones and tablets.

The MP4 container's MIME type is "video/mp4".

The WebM/VP8 Codec

The VP8 codec is an open source codec that produces relatively high quality video. Its quality is generally considered to be a bit lower than H.264 and its files a bit larger. Video encoded with this codec is stored in a WebM container, which is designed specifically for the VP8 codec.

This codec is supported by Firefox, Chrome and Opera.

The WebM container's MIME type is "video/webm".

The Ogg/Theora Codec

The Theora codec is an open source codec that produces relatively high quality video and large encoded video files. Its quality is the lowest of the three and its files the largest, but it is the most widely supported video codec and was, for a time, the primary "universal video codec" candidate for HTML5. Video encoded with this codec is usually stored in an Ogg container.

This codec is supported by Firefox, Chrome, and Opera.

The Ogg container's MIME type is "video/ogg".

The Video Codec Divide

None of the dominant video codecs are universally supported by all browsers. Internet Explorer and Safari support the MP4/H.264 video codec while Firefox, Chrome, and Opera support the WebM/VP8 and

Ogg/Theora codecs. This divide, between the proprietary MP4/H.264 codec and the open source codecs is unlikely to be bridged in the near future.

To support the widest audience, Web developers can encode their video files once for each format they wish to support and store the different formats on their servers. Using the **source** element, they can feed the correct versions to their sites' visitors with little effort.

The following example shows how to use the source element to support the widest array of users.

Using Multiple source Elements for Multiple Video Codecs

```
<video>
  <source src="video.m4a" type="video/mp4" />
  <source src="video.webm" type="video/webm" />
  <source src="video.ogv" type="video/ogg" />
</video>
```

The sources in the example are listed in descending order of quality, so that users whose browsers support the MP4/H.264 codec can get the best experience available for their platform. If storing three versions of all the video files is prohibitive, removing support for Ogg/Theora video files may be prudent, because it has the lowest quality and largest files while WebM/VP8 support is on the rise.

Web developers should keep abreast of updates regarding video codecs as support for video codecs is a quickly changing landscape.

Audio Codecs and Containers

Like video, audio is encoded and decoded by using an audio codec and stored in a container. While there is no universal audio codec or container, audio is far less contentious than video. Audio uses less data, so it taxes devices' resources less than video. Also, there are more available audio formats, so Web developers have a wider selection to choose from.

The Dominant Audio Codecs

Like the dominant video codecs, this list is not an exhaustive list of audio codecs.

The MP3 Codec

The proprietary MP3 file format contains just the encoded MP3 data without a container. In other words, MP3 is an audio codec with no container. It stores audio very efficiently and can compress the audio file to a tenth of its expanded size. It is supported by Internet Explorer, Safari, and Chrome.

The MP3 file type's MIME type is "audio/mpeg".

The AAC Codec

The proprietary AAC codec has higher quality than the MP3 codec it was designed to replace. It can exist without a container but can also be stored in a container. It is commonly stored in the MP4 container. It is supported by a very wide range of platforms, including Internet Explorer, Safari, Chrome, and most of the Microsoft, Apple, and Google smartphone devices.

When stored in a container, the AAC codec's MIME type is the container's MIME type, for example, "audio/mp4". When not stored in a container, the AAC codec's MIME type is usually "audio/aac".

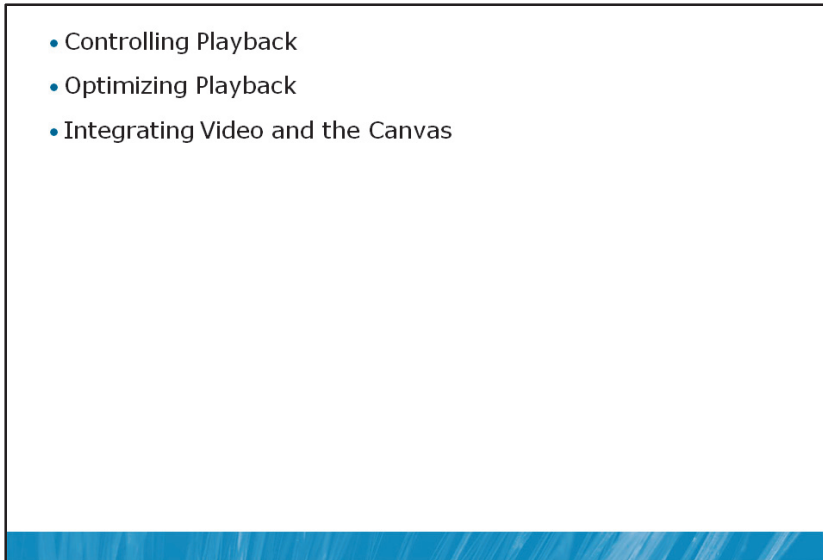
The Vorbis Codec

The open source Vorbis codec's quality is comparable with the AAC codec's quality. It is often stored in an Ogg container. The WebM container can only store Vorbis audio. It is supported by many platforms, including Firefox, Chrome, Opera, and some mobile devices.

When stored in a container, the MIME type is the container's MIME type, for example, "audio/ogg" or "audio/webm". When not stored in a container, the Vorbis codec's MIME type is usually "audio/vorbis".

Lesson 3

Controlling Multimedia with JavaScript



The preceding lesson introduced the **video** and **audio** elements, explained some of the reasoning behind them, and showed how to add them to a webpage by using declarative HTML. However, the HTML5 specifications define more than just the declarative markup. They also provide an API that can be used to manipulate the multimedia elements.

Using JavaScript to control the multimedia elements opens up many new possibilities. Web developers can simply replace the built-in controls with controls designed with the site's look and feel to full-fledged applications by using finely-controlled embedded video and audio to enrich user experience.

This lesson describes how to use JavaScript to manipulate the **video** and **audio** elements on the page.

Lesson Objectives

After completing this lesson, you will be able to:

- Control video and audio playback by using JavaScript.
- Start optimizing video and audio playback.
- Display video on the canvas.

Controlling Playback

- The **play** method starts playback
- The **pause** method pauses playback
- The **playbackRate** property changes the playback speed
- The **currentTime** property reports the current position and seeks
- The **duration** property reports the length of the video or audio
- The **buffered** property lists the currently downloaded data
- Many properties and events report status and progress

```
var video = document.getElementById("video");
video.play();
window.setTimeout(function () {
    video.pause();
}, 3000);
```

The play and pause Methods

Playing and pausing playback are the most basic means to control multimedia. The **play** method starts playing video and audio, and the **pause** method pauses video and audio playback.

The following method finds a specific video element on the page, starts playing the video, and then pauses it after 30 seconds.

Using the play and pause Methods

```
var video = document.getElementById("video");
video.play();
window.setTimeout(function() {
    video.pause();
}, 30000);
```

While the example demonstrates the use of the **play** and **pause** methods to control a **video** element, they can also be used exactly the same way with **audio** elements.



Note: There is no distinction in the Media API between “pausing” and “stopping” and the two words are used in this lesson interchangeably.

The autoplay Property

The previously covered **autoplay** attribute instructs the browser to automatically start playing the media as soon as the page loads, or enough data is downloaded. The **autoplay** property can also be set by using JavaScript. If a **video** or **audio** element is created by using JavaScript, the **autoplay** property only takes effect after the element is added to the DOM.

The following example creates a new video element that starts playing as soon as it is added to the page.

Using the autoplay Property

```
var video = document.createElement("video");
video.src = "video.m4v";
video.autoplay = true;
document.body.appendChild(video);
```



Best Practice: The **autoplay** property is very useful when video or audio has to be played automatically. However, web developers should think twice before they use the property. Excessive use of automatic playback can annoy many users. In addition, users may access the website from many different types of locations, including public locations where automatic playback might be disruptive to other people. Web developers are therefore discouraged from using the **autoplay** property and letting the user decide when to play the video or audio. If automatic playback of video is required, web developers should consider muting the video by default to avoid being disruptive. Despite the recommendation against using the **autoplay** property, it is preferred to using the **play** method because the **autoplay** property can be disabled by the browser.

The paused Property

One way to determine whether a video is playing is by examining the read-only **paused** property. When **paused** is true, the video is paused. The video may be paused for a number of reasons:

- The **Pause** button on the built-in controls was clicked.
- The **pause** method is called explicitly.
- The video has not started playing yet or has ended.

The paused property in the following example is used to determine whether to show a custom play button or pause button.

Using the paused Property

```
if(audio.paused) {
    // if the media is paused
    hidePauseButton();
    showPlayButton();
}
else {
    // if the media is playing
    hidePlayButton();
    showPauseButton();
}
```



Note: There are some cases where the media is not playing and **paused** can be false. This can happen, for example, when the video or audio is being buffered during playback and there is insufficient data to continue playing. As soon as enough data is downloaded, the media resumes playback automatically.

Media is not played automatically by default, so the **paused** property is true by default.

The play and pause Events

The Media API exposes many events that provide rich information on the state of the **video** and **audio** elements. The two events that are fired when the video is either played or paused intentionally are the **play** and **pause** events.

The **play** event is raised whenever the code or the user intentionally starts playback. This can happen, for example, when the user clicks the **Play** button on the built-in controls or when JavaScript code calls the **play** method. The **pause** event is fired when playback is paused.

The following example enhances the preceding example by toggling the play and pause buttons in response to the play and pause events.

Responding to the play and pause Events

```
audio.onpause = function(e) {
  // the media has been paused
  hidePauseButton();
  showPlayButton();
};
audio.onplay = function(e) {
  // the media is playing
  hidePlayButton();
  showPauseButton();
}
```

The playbackRate Property and the ratechange Event

The speed at which the video or audio is played is controlled with the **playbackRate** property. The **playbackRate** property is a value that is multiplied by the media's normal speed. For example, if the **playbackRate** is set to 2, the browser plays the media twice as fast.

Whenever the **playbackRate** changes, the element fires the **ratechange** event, which can be used to synchronize other events on the page. This can also be used to detect whether the user is watching fast-forwarded or slow-motion video.

The following example shows the user the video's relative playback rate.

Using the playbackRate Property and the ratechange Event

```
var video = document.getElementById("video");
var status = document.getElementById("status");
video.onratechange = function(e) {
  status.innerHTML = "The video's current rate is: " + video.playbackRate
};
```

Reporting Progress

The **currentTime** property can be used to determine the playback position. It returns a number representing the seconds from the beginning of the video or audio. The **duration** property represents the video or audio length in seconds. The two properties, together, provide some of the most basic metrics that users tend to prefer and that programmers can use to display the following visual cues:

- The current location
- The total amount of time
- Where the current location is relative to the whole as a percentage.

The two properties can be used in conjunction with the **progress** event. The **progress** event is generally fired a few times a second and can be handled to show users the current position.

The following example responds to the progress event to show users the current position as a percentage.

Reporting Progress with the progress Event

```
var video = document.getElementById("video");
video.onprogress = function() {
    var position = document.getElementById("position");
    position.innerHTML = Math.floor(video.currentTime / video.duration * 100);
};
```

Reporting progress in this way provides a very general and useful metric. This metric helps the user estimate how much time was spent watching a video or listening to an audio track, or how much time is left on the playback. However, it is not an accurate measure of how much data was downloaded, or still has to be downloaded. It also does not provide any information on how much data is buffered.

The **buffered** property returns an object containing information about the buffered data. Internally, it contains a list of time ranges that are buffered. For example, if a user has watched 15 seconds of a 60-second video and the browser has already buffered an additional 20 seconds, the **buffered** property contains a single time range starting at 0 seconds and ending at 35 seconds.

The object returned by the **buffered** property has one property of its own, **length**, which specifies how many time ranges the object contains. In other words, the object specifies how many disconnected areas of the media were buffered. Buffering multiple segments can occur, for example, when the user skips ahead. To determine what times were buffered, the object's **start** and **end** methods can be used. They both take an **index** argument that represents the time range being queried.

The following example calculates the total buffered time and shows it to the user.

Using the buffered Property

```
var video = document.getElementById("video");
var timeRanges = video.buffered;
var bufferedSeconds = 0;
for(var index = 0; index < timeRanges.length; index++) {
    bufferedSeconds = bufferedSeconds + (timeRanges.end(index) - timeRanges.start(index));
}
var message = document.getElementById("message");
message.innerHTML = bufferedSeconds + " of " + video.duration + " are currently buffered.";
```



Note: The **buffered** property is not an array and should not be dereferenced by using square brackets.



Note: Browsers may discard buffered data for various reasons. For example, if a browser estimates that a user will not seek back to the beginning of a video, the browser may discard that part. The **buffered** property only returns information about the currently buffered data, and not about the amount of data actually downloaded.

Seeking

The **currentTime** property can also be used to seek to a specific time in the playback. In other words, the playback position can be set.

The following example seeks the 30-second mark in the audio resource.

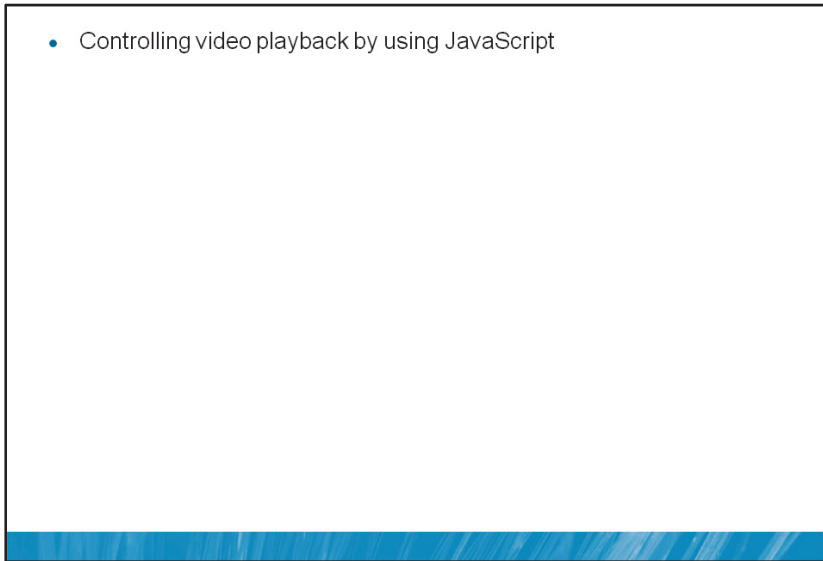
Seeking with the currentTime Property

```
var audio = document.getElementById("audio");  
audio.currentTime = 30;
```

The media element reports every stage of the seek process. This is important because the length of the process is unpredictable. When seeking starts, the media element fires the **seeking** event. When seeking finishes, the **seeked** event is raised. The Boolean **seeking** property can be checked at any time to determine if the **video** or **audio** element is currently performing a seek operation.

Not every video and audio resource supports seeking. The Boolean **seekable** property can be read to determine whether the media can be sought. In addition, even for seekable resources, some areas may not be seekable. In those cases, the browser seeks to a seekable area near the requested location.

Demonstration: Controlling Video Playback



This demonstration shows how to:

- Play and pause a video by using JavaScript.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Add the file, **D:\Mod05\Democode\Assets\trailer.m4v**, to the web application by copying or dragging the file to the new project folder in Solution Explorer.
3. Create an HTML file.
4. In the HTML file, add a **video** element with the **id** attribute and automatic playback inside the **body** element.

```
<video id="video" src="trailer.m4v" autoplay="autoplay" width="400">
</video>
```

5. Inside the **head** element, before the closing tag, add a **script** element that pauses and plays the video.

```
<script type="text/javascript">
  function toggleVideoPlayback() {
    var video = document.getElementById("video");
    if(video.paused) video.play();
    else video.pause();
  }
  function contentLoaded() {
    document.getElementById("video").addEventListener("click", toggleVideoPlayback,
false);
  }
  document.addEventListener("DOMContentLoaded", contentLoaded, false);
</script>
```

6. Save the HTML file.

7. Run the saved file.
8. After the webpage loads and the video starts playing, click the video to pause it, and click it again to resume playback.

Optimizing Playback

- Optimizing playback is important for improving performance and user experience
- The Media API provides many tools for optimizing playback
- The **readyState** property and its associated events report on the media's current state
- Different events report different reasons media stops downloading or playing

```
if (audio.readyState === audio.HAVE_ENOUGH_DATA) {  
    audio.playbackRate = 1; // play at the normal rate  
}  
else if (audio.readyState >= audio.HAVE_CURRENT_DATA) {  
    audio.playbackRate = 0.8; // slow the playback down a bit to give the  
    browser an opportunity to "catch up"  
}
```

Overview

Because video and audio can be highly bandwidth-intensive, users may encounter problems when playing media resources. The **video** and **audio** elements expose many events and properties that can help web developers fine-tune the playback and improve the user experience, whether by simply providing more detailed or intuitive status updates to the user, or by changing certain playback settings in response to changing conditions.

One issue that is traditionally difficult to solve is video slowing down as it progresses. This can be frustrating to users who watch most of the video and have difficulty getting to the end. To solve this problem, web developers have traditionally resorted to using advanced and complicated programming techniques and add-ons. To make it easier for more web developers to improve the user experience, the Media API provides a number of well-defined properties and events that can be used to account for different conditions, such as low-bandwidth and limited system resources, or conversely, systems that are more than capable of providing the best possible user experience.

This topic does not provide a comprehensive list of these properties and events, or of techniques that can be used to optimize playback on different environments. Optimizing multimedia is an advanced topic beyond the scope of this course. However, this topic does provide a small sampling of what web developers can use as a starting point.

Ready State

The readyState Property

The first property that helps inspect the status of the media is the **readyState** property. The **readyState** property is read-only and returns a single enumerated number that the programmer can use to make decisions in the code.



Best Practice: Although the **readyState** property returns a number reflecting the media's state, the media element has named constants to make the code more readable. We recommend that you use the named constants.

At any particular moment, the **readyState** property can have one of the following values:

- **HAVE_NOTHING** or 0—Indicates that there is no information. This is usually the state before the browser starts downloading any media.
- **HAVE_METADATA** or 1—Indicates that just the metadata is downloaded, so the browser knows a little bit about the media. For example, it probably knows how long the media is, whether the browser supports it, and the media's dimensions and aspect ratio.
- **HAVE_CURRENT_DATA** or 2—Indicates that the browser has enough data to display the current frame, but not enough to continue playback. This can happen, for example, when media downloads stall or when the media reaches the end (because there are no more frames).
- **HAVE_FUTURE_DATA** or 3—Indicates that the browser has enough data to display the current frame and continue playback, but not enough to play the whole video or audio resource. This is common for streamed video and audio that the browser continues to download while the media plays.
- **HAVE_ENOUGH_DATA** or 4—Indicates that the browser has all the data it needs to play the whole video or audio resource. It does not need to download the complete resource to have this state. The browser can set this state on the element if it estimates that the media can be played uninterrupted to the end of the video or audio.

The following example checks the **readyState** property to change the playback rate.

Using the **readyState** Property

```
if(audio.readyState === audio.HAVE_ENOUGH_DATA) {
    audio.playbackRate = 1; // play at the normal rate
}
else if(audio.readyState >= audio.HAVE_CURRENT_DATA) {
    audio.playbackRate = 0.8; // slow the playback down a bit to give the browser an
    opportunity to "catch up"
}
```

This code can be used on sites playing background music where the music's rate is not as important as simply having music. It can also be used on sites that show video as a strategy, to help prevent stalled videos.

readyState-Related Events

While the **readyState** property can be checked at any time, it is not always easy to know when to check the download status. The Media API provides a set of events mapping directly to the **readyState** property values (except for the **HAVE_NOTHING** value) and the events are fired when the state changes.

- The **loadedmetadata** event maps to the **HAVE_METADATA** state and is fired when the metadata of the video or audio resource is loaded.
- The **loadeddata** event maps to the **HAVE_CURRENT_DATA** state and is fired when the current frame is available.
- The **canplay** event maps to the **HAVE_FUTURE_DATA** state and is fired when there are enough buffered frames to start playing.

- The **canplaythrough** event maps to the **HAVE_ENOUGH_DATA** state and is fired when the browser estimates that the whole video or audio resource can be played uninterrupted.



Note: When the **readyState** property changes, more than one of these events might be fired. Because the **readyState** reflects increasing levels of readiness, whenever it changes to a “more ready” state, the events for all the intervening states are also fired. The same is not true when the value is reduced. So if the **readyState** is **HAVE_NOTHING**, and then it changes to **HAVE_CURRENT_DATA**, a likely scenario, both the **loadedmetadata** and the **loadeddata** events will be fired.

Stopped Events

Media download can stop for a number of reasons, including (but not limited to) users clicking the stop button, network errors, or stalled downloads. Understanding why playback stopped is important to understanding if there is a problem that needs to be solved. The Media API provides a number of events that are fired for different reasons.

- The **pause** event, already discussed, is fired when the user intentionally stops playback or when the **pause** method is called. This is not an error condition.
- The **suspend** event is fired when the browser stops buffering data intentionally, for example, because the user is not playing a partially buffered video. This is not an error condition.
- The **error** event is fired when a specific error occurs, such as when the network is disconnected unexpectedly.
- The **stalled** event is fired when a video stalls.

The stalled Event

Stalling occurs when the browser is unable to download some of the media. The **stalled** event is a good opportunity to report to the user that a problem has occurred and to provide alternatives. Recognizing poor network conditions and responding to them before the user gets too frustrated is a good strategy for keeping users engaged.

The following example informs the user that the video is stalled and highlights a direct link to the video. The user can click the link to download the video.

Responding to the stalled Event

```
var video = document.getElementById("video");
video.onstalled = function() {
    var directLink = document.getElementById("directLink");
    directLink.className = directLink.className + " highlight";
    var message = document.getElementById("message");
    message.innerText = "The browser is having difficulty downloading the video. You can download the video directly to view it on your computer.";
```

Integrating Video and the Canvas

- Video uses time, and the canvas does not
- Time loops are used to simulate time
- The **setInterval** and **setTimeout** methods drive the time loop
- The Video and Canvas APIs work well together

```
function frame() {
    var context =
document.getElementById("canvas").getContext("2d");
    var video = document.getElementById("video");
    context.drawImage(video, 0, 0);
    window.setTimeout(frame, 100);
}

frame(); // calling the function directly triggers the
first iteration
```

Overview

Generally speaking, the **video** element supports three different kinds of usage:

- Pure HTML markup
- Using the Video API
- Integrated with the Canvas API

The **video** element has enough features that are exposed through HTML to support the requirements of many types of websites. It has built-in controls, support for accessibility, multiple formats and codecs, subtitles, and synchronized playback. The Video API can be used to further control **video** elements by using JavaScript, replace the built-in controls, and provide other advanced functionality such as performance optimization. The video element can also be drawn on a canvas by using the Canvas API, where it can be combined with rich graphics and animations to create advanced graphics applications, such as advanced media players and games with video.

Controlling video when it is integrated with the canvas is different than when video is used independently. The canvas is a two-dimensional drawing surface and lacks any way to deal with time. Therefore, when video is drawn on the canvas surface, the canvas takes a snapshot of the current video image and draws just that image. However, the time factor is crucial to displaying video, so integrating the two has some unique difficulties.

This topic shows how to integrate the Video and Canvas APIs, and how to draw a playing video on the canvas.

The drawImage Method

The Canvas API exposes the **drawImage** method, which has already been introduced. Aside from ordinary single-frame images, the **drawImage** method can also draw animated images and video. Because the browser is responsible for rendering, the formats supported by the **drawImage** method are supported by the browser. So, the browser can draw on the canvas any video supported by the **video** element. In fact,

the first argument of the **drawImage** method, the **image** argument, is an **img** or **video** element, whose content the browser is expected to render directly onto the canvas surface.

The following example shows one of the overloads of the **drawImage** method used to render an **img** element on the upper-left corner of the canvas.

Drawing an **img** Element on the Canvas Surface

```
var image = document.getElementById("photo");
var canvas = document.getElementById("canvas");
var context = canvas.getContext("2d");
context.drawImage(image, 0, 0);
```

Rendering Issues

Video can be roughly described as containing a series of images shown in very quick succession to create the illusion of movement. The same idea applies to drawing video on the canvas, except that instead of the browser handling all the work, the programmer has to provide the images.

Most of the work the programmer has to perform is creating the *quick succession* of images, capturing each frame in the video *in turn* and drawing it on the surface. A simple **for** or **while** loop cannot be used because of the way the canvas is designed. Browsers do not necessarily render each image drawn onto the surface while these loops run. To optimize the process, because rendering uses many system resources, they may wait until the loop ends, and only then render the last snapshot generated by the loop.

Another problem with the **for** and **while** loops is that they run as quickly as they can. Their speed is browser- and platform-dependent, so their speed is unpredictable. However, what makes **for** and **while** loops unusable, even if browsers choose to render every iteration, is that they can be *too fast*. Video images are not just shown in quick succession; they are shown at a specific frame rate, at a metered speed. When video is played, the video player is responsible for ensuring the video is played at the correct frame rate consistently, or as close as possible. This is true for television screens, media players, and the **video** element in the browser.

Time Loops

So, a special kind of loop has to be used to simulate time. The window's **setInterval** and **setTimeout** methods are used to build the loop. Both methods have a similar signature—they accept a function and a time interval, specified as a number of milliseconds. When that time interval passes, the function argument is called. Because each frame is rendered in its own function, every iteration is always fully rendered.

The **setInterval** method is called repeatedly, iteratively, every time the interval is reached, so it serves as a natural candidate for creating time loops. However, if the rendering takes too long or the specified interval is too short, the function may not end before the next one is called. Subsequent function calls are placed in a waiting queue that consumes memory, and if every function call is slower than the interval, the queue may never empty. Therefore, the **setInterval** method is a good option when the intervals are long or the callback function short. One of its advantages is that its intervals are regular and can be used effectively to emulate a frame rate. In short, the **setInterval** method creates very simple and accurate, but risky, time loops.

The following example shows the **setInterval** method.

Creating a Time Loop with the **setInterval** Method

```
function frame() {
    // this function renders one frame and then exits
}
```

```
window.setInterval(frame, 100); // the frame function will be called 10 times every second, indefinitely
```

The **setTimeout** method is usually a better option for creating a time loop, specifically because rendering can be slow and unpredictable. The **setTimeout** method calls its callback just once, which is not enough to create a loop. To create a loop, the **setTimeout** method is called by the callback function itself, just before it exits.

The **setTimeout** method creates safer, but slightly less accurate, time loops, as the following example shows.

Creating a Time Loop with the **setTimeout** Method

```
function frame() {
  // this function renders one frame and then triggers another iteration
  window.setTimeout(frame, 100);
}
frame(); // calling the function directly triggers the first iteration
```

Rendering the Frame

After a loop is established, the frame can be rendered. Frames are rendered by calling one of the **drawImage** methods overloads. However, the **drawImage** method provides no way to specify which frame to draw. It always draws the current frame.

This time loop renders the current video frame in every iteration, as the following example shows.

Rendering the Current Video Frame in a Time Loop

```
function frame() {
  var context = document.getElementById("canvas").getContext("2d");
  var video = document.getElementById("video");
  context.drawImage(video, 0, 0);
  window.setTimeout(frame, 100);
}
frame(); // calling the function directly triggers the first iteration
```

Playing the Video

Fortunately, the Video API can be used to play the video and advance the frames. In other words, the video is played normally—manually by the user, by setting **autoplay** to true, or by calling the **play** method of the **video** element. The **drawImage** method renders whichever frames happens to be the current frame.

To copy the video to the canvas so it appears to run smoothly on the canvas, the time loop has to use a sufficiently high speed. Experimentation is the best way to determine the ideal speed, but 20 or 30 millisecond intervals are commonly used.

The following example shows a complete time loop implementation that starts playing the video as soon as the page is loaded, by using jQuery to assist.

A Complete Video-Rendering Time Loop

```
$(function() {
  var video = $("#video")[0];
  video.preload = "auto";
  video.onplaying = frame;
  video.play();
});
```

```
function frame() {  
    var context = $("canvas")[0].getContext("2d");  
    var video = $("video")[0];  
    if(video.playing) {  
        context.drawImage(video, 0, 0);  
        window.setTimeout(frame, 20);  
    }  
}
```

As soon as the page finishes loading, jQuery runs the code in the anonymous function. The function finds the first **video** element with the **id** "video", and instructs the browser to preload the video aggressively (using the "auto" value). When the video starts playing, the **playing** event is fired and calls the **frame** function that starts the loop.

Using the Video API and the Canvas API together is not limited to just starting playback. The time loop is an effective mechanism to draw almost any kind of moving graphic on the canvas. The APIs are designed to coexist well together.

Lab: Getting Started with Graphics and Multimedia Elements

- Adding Video to the Details Page
- Creating a Slideshow by Using the Canvas

Logon Information

Virtual Machine	Virtual machine: SEA-DEV
User Name	User name: Admin
Password	Password: Pa\$\$w0rd

Estimated Time: 60 minutes

Scenario

The Contoso, Ltd. management has decided that the online store should have a more interactive game details page. The game details page, which currently contains only images, will now also contain trailers of the games. The trailers and images should be presented in a slideshow that cycles through the available images and trailers.

Objectives

After completing this lab, you will be able to:

- Integrate a **canvas** element into the game details page.
- Integrate a **video** element into the game details page.

Exercise 1: Adding Video to the Details Page

Scenario

Before implementing the slideshow, management wishes to see a proof of concept (POC) demonstrating the use of video on the game details page. The video should play automatically and should be compatible with multiple browsers.

A sample video in multiple formats is provided in the **Content\Videos\Games\1** folder in the project.

► Task 1: Replace the large image with a new video element.

1. Open the Starter solution.
2. Open the game details view.
3. Remove the large image.
4. Add a new **video** element.
5. Set the **video** element's size.
6. Test the application.

► Task 2: Add support for multiple video sources.

1. Replace the **src** attribute with multiple **source** elements.

► Task 3: Set the video element's behavior.

1. Set the video to play automatically.
2. Mute the video to avoid annoying users.
3. Set the video to loop automatically.
4. Test the application.

► Task 4: Add fallback behavior.

1. Add a poster.
2. Add fallback content.

Results: A Game Details webpage with a **video** element is displayed.

Exercise 2: Creating a Slideshow by Using the Canvas

Scenario

Management has approved the Proof of Concept (POC) and asked you to implement the full slideshow. The slideshow should support video trailers, switch slides automatically, and let the user start and stop video playback. Older browsers that do not support the **canvas** element should be able to see the large image.

► Task 1: Create a canvas element for the slideshow.

1. Add a new **canvas** element.
2. Set the **canvas** element's size.

► Task 2: Add fallback behavior.

1. Wrap the static slideshow images and video into the **canvas** element.

► Task 3: Initialize the slideshow.

1. Open the JavaScript file.
2. Store a reference to the canvas's 2D context.
3. Create **img** and **video** elements based on the dynamically-retrieved **game** object's data.

► Task 4: Draw outlines for the slides on the canvas.

1. Draw an outline for the main slide.
2. Draw an outline for each thumbnail slide.

► Task 5: Draw the thumbnail slides on the canvas.

1. Draw **img** elements on the canvas.
2. Draw **video** elements on the canvas.
3. Add a time loop to redraw the video continuously as the playback progresses.
4. Use the Video API to ensure that only playing videos are drawn.

► Task 6: Switch the main slide at regular intervals.

1. Change the current slide automatically at regular intervals.
2. Draw the current slide as the main slide.

► Task 7: Toggle video playback when the user clicks a slide.

1. Play or pause the video when the user clicks the main slide.
2. Draw play and pause symbols.
3. Test the application.

Results: A Game Details webpage is displayed with a **canvas** element that plays a video.

Module Review and Takeaways

This module introduced some of the most exciting features in HTML5. The multimedia elements—**video** and **audio**—are not just excellent replacements for legacy add-ons. They are designed for compatibility with different types of browsers, are not locked to one media format, have many accessibility features, and perform very well. In addition, they have extensive APIs that programmers can use to control multimedia.

The canvas is also more than just a replacement for add-ons. It can achieve excellent performance by utilizing hardware assistance, has a rich API for drawing two-dimensional graphics, and can draw images and play videos. Programming graphics can be difficult, but the Canvas API simplifies many of the tasks associated with graphics programming, so that web developers can benefit from its use, regardless of their experience.

Review Questions

When using JavaScript to control video playback, what considerations affect whether a **video** element should be used alone, or played on a canvas?

Module 6

Creating Advanced Graphics

Contents:

Module Overview	6-2
Lesson 1: Drawing with SVG	6-3
Lesson 2: Animation	6-22
Lesson 3: Choosing Between SVG and Canvas	6-39
Lab: Creating Advanced Graphics	6-43
Module Review and Takeaways	6-46

Module Overview

- Drawing with SVG
- Animation
- Choosing Between SVG and Canvas

HTML5 has two different graphics models that are already widely supported: Scalar Vector Graphics (SVG) and the Canvas. Although the two specifications are very different, some of the concepts used with the Canvas also apply when using SVG.

SVG is an XML-based markup language that defines a Document Object Model (DOM), like XHTML, which is an XML variant of HTML. The **svg** element is the SVG document's root element, just as the **html** element is the HTML document's root element. More significantly, SVG is a *declarative language* that describes drawings, just as HTML describes a webpage. The SVG or HTML document contains data, and the browser is responsible for rendering that data.

SVG is older and more developed than the Canvas, but SVG has only recently grown popular, following the arrival of the Canvas in HTML5. Both can be used to create advanced graphics. This module teaches how to add animation.

Both the Canvas and SVG are general-purpose graphics libraries and can be used for almost any task that requires graphics, which can make it difficult to decide which library to use for which type of task. This module discusses some of the differences and provides guidance on how to choose one over the other.

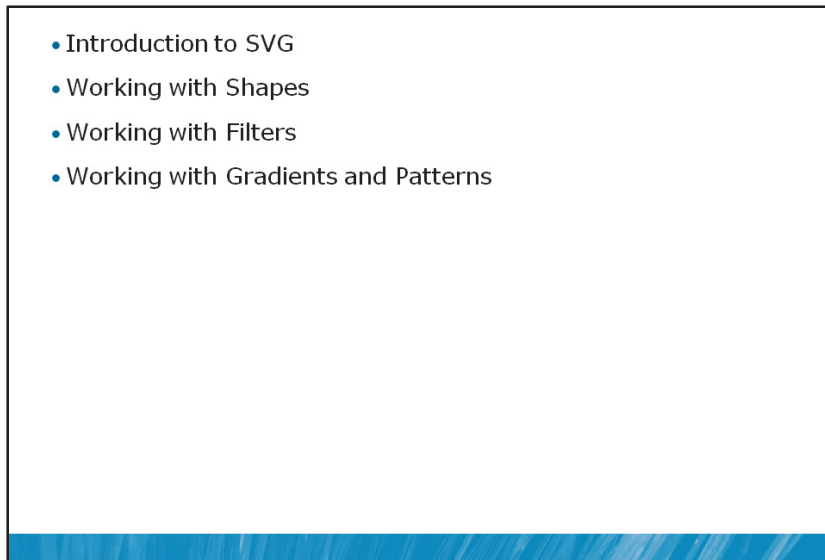
Objectives

After completing this module, you will be able to:

- Create basic graphics with SVG.
- Add animation to SVG and Canvas graphics.
- Choose between SVG and Canvas.

Lesson 1

Drawing with SVG



Scalable Vector Graphics (SVG) is another graphics model that can be used with HTML. It is very different from the Canvas, which is pixel-based. The use of SVG was first recommended in 2001, but still wasn't popular before the emergence of HTML5. Now, it is gaining a lot of attention as a complementary tool to the Canvas, and is also sometimes used on its own to create graphics.

In this lesson, you will learn about SVG and how to work with it in HTML to draw graphics. You will learn how to create basic shapes such as rectangles and circles, and how to use more advanced graphics options such as filters and patterns.

Lesson Objectives

After completing this lesson, you will be able to:

- Use SVG to create graphics.
- Apply SVG shapes.
- Apply SVG filters.
- Apply SVG gradients and patterns.

Introduction to SVG

- SVG stands for *Scalable Vector Graphics*
- Defines graphics by using an XML model; embedded in HTML by using an **<svg>** tag
- Is part of the DOM
- Became a recommendation of W3C in 2001
- The specification was edited in 2011

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg"></svg>
```

In 2001, the SVG 1.0 specification was made into a recommendation by W3C. In 2011, W3C introduced a second edition of SVG 1.1, which was more HTML5-oriented. It was uncommon for developers to use previous versions of SVG because of a lack of browser support and difficult specifications leading to the widespread use of plug-ins such as Flash for graphics creation. This scenario is changing today.

Scalable Vector Graphics (SVG) is a graphics model that leaves the graphics quality of an element unaffected by changes in size or zoom. When you use SVG, you define the graphics with XML and the **<svg>** tag.

The following example shows an empty svg element.

SVG Element

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
</svg>
```

As you can see, the **svg** element includes a **version** attribute with a value of 1.1, indicating that the specific 1.1 version of the specifications is being used. The element also includes an XML namespace (**xmlns**) that holds a link to the SVG schema.

The **svg** element can be embedded in HTML like any other DOM element, and can be manipulated by using JavaScript, like any other DOM element.

The following example shows the SVG element embedded inside a simple webpage.

SVG Embedded in HTML

```
<!DOCTYPE html>
<html>
<head>
  <title>SVG Example</title>
</head>
<body>
  <div>
    <svg version="1.1" xmlns="http://www.w3.org/2000/svg">
```

```
        </svg>  
    </div>  
</body>  
</html>
```

SVG can be used to replace image resources, which are downloaded by using HTTP requests. This can decrease the amount of requests to the server, resulting in better performance. SVG is used to create graphics that can be printed at a high quality regardless of the resolution. These graphics can be zoomed in and out without losing any quality. All these advantages make SVG a very powerful graphics model for use in HTML.

Working with Shapes

- Inside the `<svg>` tag, you can embed a number of shape elements
 - Rectangles
 - Circles
 - Ellipses
 - Polygons
 - Lines
 - Polylines
 - Paths
 - Text
- Shapes can be styled, grouped, and can have transformations.

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
<g id="group1" fill="blue">
  <rect x="30" y="30" width="100" height="60" />
  <text x="50" y="50" font-family="Arial" font-size="30">Hello SVG</text>
</g>
</svg>
```

SVG can embed shape elements such as rectangles and circles. Each shape has its own tag and set of possible attributes. Shape elements can be combined to create more advanced shapes and graphics.

Rectangles

Rectangles are created with the **rect** element. The main rectangle attributes are:

- x**—The coordinate along the x-axis inside the svg surface. The attribute default value is 0.
- y**—The coordinate along the y-axis inside the svg surface. The attribute default value is 0.
- width**—The width of the rectangle. The default unit for the attribute is in pixels.
- height**—The height of the rectangle. The default unit for the attribute is in pixels.
- rx**—The x-axis radius of the corner ellipse to create rounded corners to the rectangle.
- ry**—The y-axis radius of the corner ellipse to create rounded corners to the rectangle.

The following example shows a simple rectangle.

The rect Element

```
<rect x="30" y="30" width="100" height="60" />
```



FIGURE 6.1: THE RECTANGLE ELEMENT

Circles

Circles are created with the **circle** element. The main circle attributes are:

- **cx**—The x-axis coordinates of the center of the circle. The default value is 0.
- **cy**—The y-axis coordinates of the center of the circle. The default value is 0.
- **r**—The radius of the circle. If this attribute is set to 0, the element will not be rendered. If the value is negative, it will raise a rendering error.

The following example shows a simple circle.

The circle Element

```
<circle cx="50" cy="30" r="20"/>
```



FIGURE 6.2: THE CIRCLE ELEMENT

Ellipses

Ellipses are created with the **ellipse** element. The main ellipse attributes are:

- **cx**—The x-axis coordinates of the center of the ellipse. The default value is 0.
- **cy**—The y-axis coordinates of the center of the ellipse. The default value is 0.
- **rx**—The x-axis radius of the ellipse. If this value is 0, the shape will not be rendered.
- **ry**—The y-axis radius of the ellipse. If this value is 0, the shape will not be rendered.

The arc of the ellipse begins at the 3 o'clock point and progresses to the 9 o'clock point. The arc also depends on any transformations that may have been applied to the shape. Transformations are discussed later in this topic.

The following example shows a simple ellipse.

The ellipse Element

```
<ellipse cx="400" cy="200" rx="350" ry="150" />
```

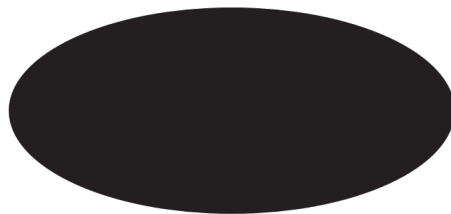


FIGURE 6.3: THE ELLIPSE ELEMENT

Polygons

Polygons are closed shapes constructed from connected lines. They are created with the **polygon** element. The main attribute of a polygon is **points**, whose value is a list of coordinate points that the polygon will comprise. The number of points must be even or a render error will be raised. Each point is separated by a comma, and is made up of two numbers, representing the x and y coordinates, separated by a space. If only one value is supplied, it is used for both coordinates.

The following example shows a simple polygon.

The polygon Element

```
<polygon points="15, 5, 100 8,6 150" />
```



FIGURE 6.4: THE POLYGON ELEMENT

Lines

Lines start at one point and end at another point. They are created with the **line** element. This element makes use of the following main attributes:

- **x1**—The x coordinate of the start of the line. The default value is 0.
- **y1**—The y coordinate of the start of the line. The default value is 0.
- **x2**—The x coordinate of the end of the line. The default value is 0.
- **y2**—The y coordinate of the end of the line. The default value is 0.

The following example shows a simple red line.

The line Element

```
<line x1="100" y1="300" x2="300" y2="100" stroke="red" />
```



FIGURE 6.5: THE LINE ELEMENT



Note: The stroke attribute will be learned later in this topic. Without it, the line will not be drawn.

Polylines

Polylines describe a set of connected line segments. They are represented by the **polyline** element. They differ from polygons in that polygons are closed shapes, while polylines can be open shapes. As with polygons, the main attribute of a polyline is **points**, which behaves much the same as it does in the **polygon** element.

The following example shows a simple polyline.

The polyline Element

```
<polyline points="50,375 150,375 150,325 250,325 250,375 350" />
```



FIGURE 6.6: THE POLYLINE ELEMENT

Paths

Paths, as their name indicates, describe a path. They are created with the **path** element. Paths have two main attributes:

- **d**—The path data. Defines the outline of the shape.
- **pathLength**—The total length of the path.

The path data includes commands that instruct the browser how to create the path. These commands are represented by letters, followed by the x and y coordinates at which to apply the command. An uppercase command letter indicates absolute positioning, while lowercase letters indicate relative positioning.

The following list describes the available commands:

- **M**—moveto
- **L**—lineto
- **H**—horizontal lineto
- **V**—vertical lineto
- **C**—curveto
- **S**—smooth curveto
- **Q**—quadratic Bézier curve
- **T**—smooth quadratic Bézier curveto

- A—elliptical arc
- Z—closepath

The following example shows how to draw a path.

The path Element

```
<path class="SamplePath" d="M100,200 C100, 100 250, 100 250, 200 S400, 300 400, 200" />
```



FIGURE 6.7: THE PATH ELEMENT

Text

To render text in SVG, you must use the **text** element. Line breaks and word wrapping are not supported in SVG and must be created manually by the developer. This means that you must break lines of text on your own by using more than one text element. The text is written between the starting tag and the closing tag.

The following example shows how to use the text element.

The text Element

```
<text>Hello SVG</text>
```

The **text** element has the following main attributes:

- **x**—The x coordinate of the text from the upper left corner of the svg element.
- **y**—The y coordinate of the text from the upper left corner of the svg element.
- **dx**—A list of lengths (separated by commas) that describe the shift in the position of the text in the x-axis.
- **dy**—A list of lengths (separated by commas) that describe the shift in the position of the text in the y-axis.
- **rotate**—A list of numbers that set the rotation of the text's glyphs, which are the building blocks of the text.

The element also has the ability to use all the properties available in the **font** CSS family, such as **font-family**, which indicates which font to use, or **font-size**, which sets the size of the text.

The following example shows a more styled text element.

Text Element with Font Attributes

```
<text x="50" y="50" font-family="Arial" font-size="30" >Hello SVG</text>
```

Adding Style to Shapes

SVG offers many style attributes that can be applied to the shape elements. One of the basic styles is the stroke, which contains the following attributes:

- **stroke**—Defines the color of the line, the text, or the outline of an element.
- **stroke-width**—Defines the thickness of the line, the text, or the outline of an element.
- **stroke-linecap**—Defines a style for the end of an open path. These styles include square, round, and butt.
- **stroke-dasharray**—Used to create dashed lines. The attribute gets a list of comma and/or white space-separated lengths and percentages. Every item in the list specifies the lengths of alternating dashes and gaps.

The following example shows how to use the stroke attributes.

Stroke Attributes

```
<path stroke="blue" stroke-width="4" stroke-linecap="round" stroke-dasharray="10,10"
d="M5 20 1215 0" />
```

The fill attributes determine the fill of the elements. The main fill attributes are:

- **fill**—Paints the interior of the element. Open paths and polylines that use the fill attribute will be rendered much like closed paths or polygons. If this attribute is not used, the shapes will be filled in black.
- **fill-opacity**—Determines the opacity of the fill. Can receive values in the range of 0 to 1. The default value is 1, which means fully opaque.

The following example shows the use of the fill attributes in a rectangle.

Fill Attributes

```
<rect x="30" y="30" width="100" height="60" fill="red" fill-opacity="0.3" />
```



FIGURE 6.10: FILL ATTRIBUTE

There are also two attributes that control the visibility of SVG elements: **display** and **visibility**. The **display** attribute can be set on a container element, as opposed to **visibility**, which can only be set on single elements. When **display** is set to "none", the element it was applied to will not be a part of the rendering tree. If **visibility** is set to "hidden", the element is rendered and takes up space even though it is not visible.



Additional Reading: There are a lot of other attributes that can be set on elements to change their appearance. You are encouraged to read about these other options at the following link: [Attributes](#)

SVG supports CSS styling. You can set styles such as stroke and fill by using CSS syntax, whether inline in the element, inline in the SVG, or in external CSS files referenced by the webpage. You can also use the class attribute for CSS as you would in any HTML element.

The following example shows how to draw a rectangle with an inline style.

CSS Syntax

```
<rect x="200" y="100" width="600" height="300" style="fill: red; stroke: blue; stroke-width: 3"/>
```



Best Practice: It is preferable to set SVG style such as stroke or fill by using their attributes, instead of style properties (as in the previous example). This might help to manipulate them later by using JavaScript.

Element Grouping

SVG makes use of the **g** element, which is used as a container of elements. **g** elements can be nested inside other **g** elements to create a hierarchy of grouping. A style that is applied to a **g** element is applied to all the other elements in the group.

The following example shows how you can use a **g** element to group two rectangles.

The **g** Element

```
<g id="group1" fill="blue">
  <rect x="10" y="10" width="10" height="10"/>
  <rect x="100" y="100" width="10" height="10"/>
</g>
```

Using Transformations in SVG

As in CSS3, SVG has the ability to apply transformations to SVG elements. This is accomplished by using the **transform** attribute, which can be set on an SVG grouping element or on any of the SVG elements that you learned in this topic. It receives as its value a list of transformations, separated by spaces, that are applied in the order they are given.

There are a few transformation definitions that resemble the CSS3 transform options:

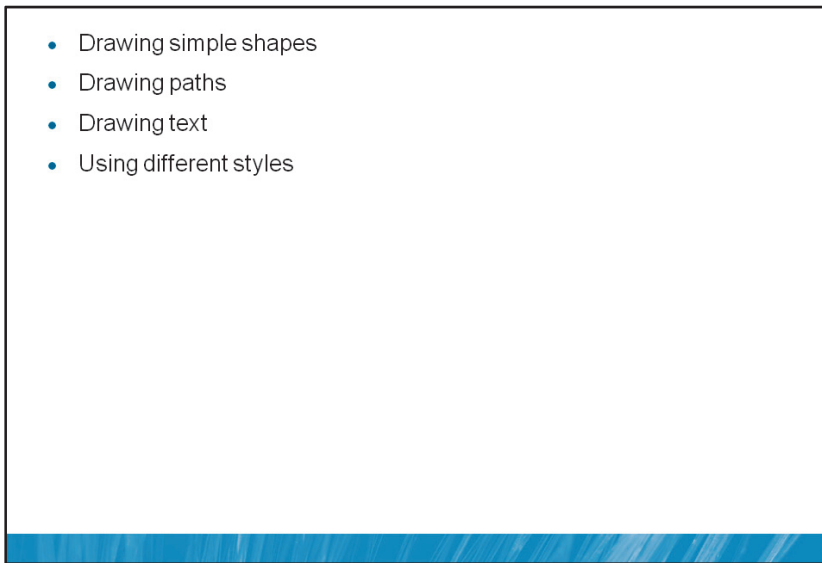
- `matrix(a, b, c, d, e, f)`—Applies the matrix transformation [a b c d e f] to the element.
- `translate(tx, ty)`—Moves the element by tx and ty. If ty is not provided, it defaults to 0.
- `scale(sx, sy)`—Scales the element by sx and sy. If sy is not provided, it defaults to the value given for sx.
- `rotate(rotate-angle, x, y)`—Rotates the element by the given angle in degrees. Rotate can also receive, as two optional parameters, the x and y coordinates of the point around which the element will be rotated.
- `skewX(skew-angle)`—Skews the element along the x-axis by the given angle.
- `skewY(skew-angle)`—Skews the element along the y-axis by the given angle.

The following examples show how to use the transform attribute.

The transform Attribute

```
<rect x="100" y="100" width="20" height="20" transform="scale(4)"/>
<g transform="translate(10, 20) scale(3) rotate(90) translate(5, 5)">
  <!-- graphics elements that the transform applies to -->
</g>
```


Demonstration: Drawing Shapes



This demonstration shows how to:

- Draw a rectangle.
- Draw a circle.
- Draw a path.
- Draw text.
- Apply styles.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file, add an **svg** element with the correct version and schema inside the **body** element.

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
</svg>
```

4. In the **svg** element, add a **rect** element.

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
  <rect x="20" y="30" width="150" height="40" rx="5" ry="5" fill="orange" />
</svg>
```

5. Before the **svg** element's closing tag, add a **circle** element.

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
  ...
  <circle cx="250" cy="50" r="40" fill="blue" />
</svg>
```

6. Before the **svg** element's closing tag, add a **path** element.

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
...
<path d="M 20,200 C 20,100 170,100 170,200 S 320,300 320,200" fill="red"
stroke="green" stroke-width="3" />
</svg>
```

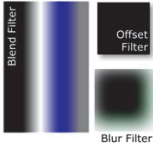
7. Before the **svg** element's closing tag, add a **text** element.

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
...
<text x="160" y="130" font-family="Arial" font-size="36" fill="black" stroke="gray"
stroke-width="2">Hello SVG</text>
</svg>
```

8. Save the HTML file.
9. Run the saved file.

Working with Filters

- SVG uses the **defs** element to define filters and other special SVG elements
- Filters that can be used with SVG include:
 - Blur filter
 - Offset filter
 - Blend filter



```

<svg version="1.1" xmlns="http://www.w3.org/2000/svg" >
  <defs>
    <filter id="f1" x="0" y="0" width="30" height="20">
      <feGaussianBlur in="SourceGraphic" stdDeviation="10" />
    </filter>
  </defs>
</svg>

```

Filters can help apply better graphical effects to elements and element containers. Using filters allows graphic designers and developers to decrease the dependency on images in webpages to achieve effects like blur or shadow. This decreases the amount of resources that the webpage requires and lowers the amount of HTTP requests.

A filter is a series of graphical operations that are applied to an SVG element, modifying its representation. To use filters, you first need to learn about the **defs** element. This element contains the definitions of special SVG elements, including filters. In the next topic, you will learn about other definitions that the **defs** element can contain. Inside the **defs** element, you can use the **filter** element to define an SVG filter. Filter elements have **x**, **y**, **width**, and **height** attributes that define a rectangular region inside which the filter is applied.

The following example defines a filter element.

Filter Element

```

<svg version="1.1" xmlns="http://www.w3.org/2000/svg" >
  <defs>
    <filter id="f1" x="0" y="0" width="30" height="20">
    </filter>
  </defs>
</svg>

```



Note: You must provide an **id** for the filter element, which will be used later by the elements that use the filter.

SVG provides a large number of filter effects. Their most common attributes are:

- **in**—The input for the filter. Can receive values such as **SourceGraphic**, **SourceAlpha**, **BackgroundImage**, **BackgroundAlpha**, **FillPaint**, or **StrokePaint**:
 - SourceGraphic**—The graphic elements that were the original input of the filter element.
 - SourceAlpha**—The alpha channel of the graphic elements that were the original input of the filter

element.

BackgroundImage—The image that was under the filter element when the filter was invoked.

BackgroundAlpha—The alpha channel of the image that was under the filter element when the filter was invoked.

FillPaint—The fill property on the target element for the filter effect.

StrokePaint—The stroke property on the target element for the filter effect.

The attributes default to the **in** value of the previous filter, or to **SourceGraphic** if there is no previous filter.

- **x, y, width, and height**—Very similar to the filter element's attributes.
- **result**—The graphics that result from processing the filter can be referenced by an **in** attribute on a subsequent filter in the same filter element. If there is no value and the next filter does not supply an **in** attribute, the next filter will use the result of this filter as input.

The following filters can be used in SVG:

feGaussianBlur—Creates a blur effect. Uses the **stdDeviation** attribute, which defines the amount of blur to set on the element.

The following example defines a **feGaussianBlur**.

feGaussianBlur Filter Definition

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" >
  <defs>
    <filter id="f1" x="0" y="0">
      <feGaussianBlur in="SourceGraphic" stdDeviation="10" />
    </filter>
  </defs>
</svg>
```

feOffset—Moves an element to another location in the xy plane. The **dx** and **dy** attributes move the element in the x-axis and the y-axis, respectively.

The following example shows how to use the **feOffset** filter.

feOffset Filter

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg" >
  <defs>
    <filter id="f1" x="0" y="0">
      <feOffset in="SourceGraphic" dx="20" dy="20" />
    </filter>
  </defs>
</svg>
```

feBlend—Blends two elements together by using a method selected in the **mode** attribute. This attribute can accept values such as normal, multiply, screen, darken, and lighten. The **in2** attribute acts as a second **in** attribute for this filter, and it is used to indicate the second element for the blend.

The following example composes the previous **feOffset** filter with the **feBlend** filter.

feBlend Filter

```
<svg xmlns="http://www.w3.org/2000/svg" version="1.1">
  <defs>
    <filter id="f1" x="0" y="0">
      <feoffset result="off0ut" in="SourceGraphic" dx="10" dy="10" />
```

```

        <feblend in="SourceGraphic" in2="offOut" mode="normal" />
      </filter>
    </defs>
  </svg>

```

There are many other filters that can be used in SVG, and using them in conjunction with each other can create very sophisticated graphics.

To apply a filter to an SVG element, you must use its id.

The following example shows how to use a filter.

A Filter Set on an Element

```

<svg version="1.1" xmlns="http://www.w3.org/2000/svg" >
  <defs>
    <filter id="f1" x="0" y="0">
      <feGaussianBlur in="SourceGraphic" stdDeviation="10" />
    </filter>
  </defs>
  <rect width="120" height="120" fill="blue" filter="url(#f1)" />
</svg>

```



FIGURE 6.11: A FILTER SET ON AN ELEMENT



Additional Reading: For additional reading about other filters, go to the following link:
[Filters](#)

Working with Gradients and Patterns

- SVG allows for linear and radial gradients.
- SVG provides the ability to define a graphic pattern that can be applied to other elements.
- The gradient and pattern definitions are put inside the **defs** element, like filters.

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
  <defs>
    <linearGradient id="g1">
      <stop offset="20%" stop-color="green" />
      <stop offset="80%" stop-color="blue" />
    </linearGradient>
  </defs>
  <rect fill="url(#g1)" x="20" y="20" width="200" height="100"/>
</svg>
```

As in the Canvas API, SVG includes the ability to use linear and radial gradients. As with filters, gradients must be defined inside the **defs** element and must be identified by an **id**. There are two gradient elements: **linearGradient** and **radialGradient**. Inside these elements, you can nest a set of **stop** elements, defining each color to be used in the gradient. The **stop** element makes use of the **stop-color** attribute to define the stop's color and the **offset** attribute to indicate the percentage of the gradient that applies to that color.

The following example shows how to use a linear gradient.

Linear Gradient

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
  <defs>
    <linearGradient id="g1">
      <stop offset="20%" stop-color="green" />
      <stop offset="80%" stop-color="blue" />
    </linearGradient>
  </defs>
  <rect fill="url(#g1)" x="20" y="20" width="200" height="100"/>
</svg>
```

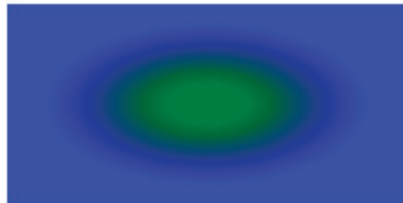


FIGURE 6.12: LINEAR GRADIENT

The following example uses the preceding code to show a radial gradient.

Radial Gradient

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
  <defs>
    <radialGradient id="g1">
      <stop offset="20%" stop-color="green" />
      <stop offset="80%" stop-color="blue" />
    </radialGradient>
  </defs>
  <rect fill="url(#g1)" x="20" y="20" width="200" height="100"/>
</svg>
```

**FIGURE 6.13: RADIAL GRADIENT**

Another definition element in SVG is the pattern element. Like filters and gradients, it must be placed in the defs element. The pattern element creates a graphical pattern that can later be applied to SVG elements. This makes it possible to fill elements with something other than a solid color or a gradient. Inside the pattern element, you can nest all shapes you learned in this lesson.

The pattern's main attributes are:

patternUnits—Define the coordinate system for the x, y, width, and height attributes. You can get two values: *userSpaceOnUse* and *objectBoundingBox*. The default value is *objectBoundingBox*. The *userSpaceOnUse* means that the current user coordinate system is applied. The *objectBoundingBox* means that the bounding box of the element that use the pattern is taken as the current coordinate system.

x, y, width, and height—These represent the location of the pattern on the x-axis and y-axis, and the pattern width and height.

The following example shows how to use a pattern.

Pattern Element

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
  <defs>
    <pattern id="p1" patternUnits="userSpaceOnUse" width="150" height="100">
      <circle cx="50" cy="50" r="10" fill="yellow" stroke="blue" />
      <rect x="100" y="0" width="50" height="50" fill="blue" stroke="black" />
    </pattern>
  </defs>
  <ellipse fill="url(#p1)" cx="200" cy="200" rx="150" ry="150" />
</svg>
```

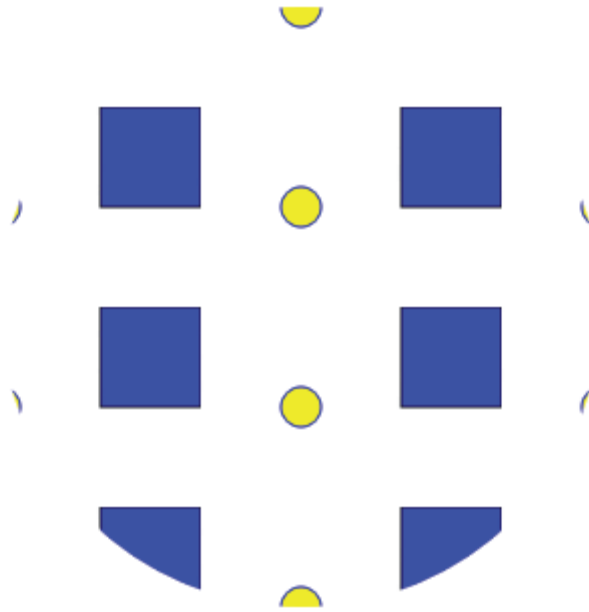


FIGURE 6.14: PATTERN

Lesson 2

Animation

- Creating SVG Animation by Using Markup
- SVG Animation by Using JavaScript
- Canvas Animation by Using JavaScript

Both SVG and the Canvas can draw simple and complex graphics, support JavaScript, offer compelling alternatives to proprietary add-ons, are standardized by the W3C and are supported by the most current browsers.

As previously demonstrated, both can be used to create static drawings. SVG and the Canvas also support animation, drawings that depict motion or other changes over time. By using JavaScript with the Canvas or with SVG, or by using SVG markup, any web developer can add animation to graphic applications. Games, slideshows, presentations, charts, and graphs, to name just a few, can all be animated with SVG or the Canvas.

This lesson shows how to add animation to SVG and Canvas drawings.

Lesson Objectives

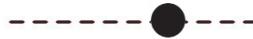
After completing this lesson, you will be able to:

- Create SVG animations by using only SVG markup.
- Create SVG animations by using JavaScript.
- Create Canvas animations by using JavaScript.

Creating SVG Animation by Using Markup

- SVG supports declarative animations by using markup
- SVG has a flexible unified syntax for time
- SVG has multiple animation elements:
 - The **set** element changes a single value at once
 - The **animate** element changes a single value gradually
 - The **animateMotion** element moves a shape along a path
 - The **animateTransform** element applies a transformation to a shape

```
<circle r="30" cx="50" cy="50" stroke="black" fill="darkgreen">
  <animate attributeName="cx" to="150" begin="click" dur="5" />
</circle>
```



Overview

Creating SVG animation by using markup is very different from using a *procedural language* such as JavaScript to execute commands (statements) that instruct the browser precisely what to do, step-by-step. SVG and HTML only tell the browser what the document contains, and leave the implementation (the step-by-step details) to the browser. As an example, a **rect** element in SVG instructs the browser that a rectangle must be drawn, and the browser determines how and when to draw that rectangle, whereas using the Canvas API's **fillRect** method sends an immediate instruction to draw the rectangle directly on the canvas. The difference may seem subtle, but it has important implications.

The most significant implication is that the canvas uses immediate-mode graphics, but SVG uses retained-mode graphics. Relying on immediate-mode graphics means that the canvas only remembers the basic bitmap data—the actual drawn array of pixels and colors—and not the series of instructions that produced them. The canvas has no data structure that the web developer can use to determine if a certain point is part of a button or some other shape. Typically, the web developer has to store this information in a custom data structure, usually by using JavaScript objects, and implement some form of custom hit-testing mechanism to determine which pixels are associated with which objects.

SVG, on the other hand, uses retained-mode graphics, so the browser is responsible not only for rendering the SVG document, but also for keeping the document and the drawing synchronized, without any JavaScript. Put simply, the SVG document *is* the drawing's data. When the document or the drawing changes, so does the other. Even when JavaScript is used to modify an SVG drawing, it actually modifies the SVG document and then the browser updates the drawing.

The SVG reliance on a document model and retained-mode graphics means that SVG drawings are more abstract than the low-level canvas drawings. The browser takes care of a lot of details. This has also proven useful for animations. In addition to the elements representing shapes, groups, and filters, SVG has a number of elements that declare animations. The animation elements describe the animations and the browser is responsible for implementing them. For example, the animation elements describe the animation type, how long the animation should take, and what the animated elements look like before and after the animation is performed. They do not tell the browser how quickly the animation must be

refreshed or how many frames per second must be drawn. Those are low-level details managed by the browser.

Time

Time drives an animation. Time is the main difference between a static image and an animation. Web developers specify when animations should run, how long they should last, whether multiple animations run in parallel or one after another, and whether user actions, such as mouse and keyboard events, trigger certain animations. To give web developers this kind of control over time in their animations by using only markup (without resorting to JavaScript), SVG provides a unified syntax for specifying time.

Every animation element has two optional attributes—**begin** and **end**—that specify when an animation starts and finishes. The attribute values use the unified syntax to determine exactly when an animation runs. The unified syntax is very flexible, and it supports specifying time in multiple ways. It can be used, for example, to specify that an animation runs from three seconds after the SVG document loads, or when a user clicks some element, or five seconds before another animation ends.

Clock Time

Clock time refers to simple time the way it is normally used (for example, “three minutes” or “five o’clock”) and can be specified in a number of different ways. All of the following examples use the **begin** attribute, although the **end** attribute can also be used in the examples.

- A simple number, on its own, indicates the number of seconds. For example:
 - `begin="5"` = five seconds
 - `begin="10.5"` = ten seconds and five hundred milliseconds
- A simple number followed by “h”, “min”, “s”, or “ms” specifies the number of hours, minutes, seconds, or milliseconds, respectively. For example:
 - `begin="10.5s"` = ten seconds and five hundred milliseconds
 - `begin="5.5min"` = five minutes and thirty seconds
 - `begin="1h"` = one hour
- Standard time notation by using colons (":") indicates a specific number of hours, minutes, and seconds. The hours, minutes, and seconds must each be specified by using two digits, padding an extra zero on the left if necessary, so “three minutes”, for example, is written as “03”. The hours are optional, and the seconds can be followed by a dot (".") and the number of milliseconds. For example:
 - `begin="05:30"` = five minutes and thirty seconds
 - `begin="10:20:30.400"` = ten hours, twenty minutes, thirty seconds and four hundred milliseconds

Triggers

Different trigger types can be used to start or stop animations. The following list describes the supported triggers. To start and stop an animation when:

- *Another* animation starts or stops, the other animation’s ID is specified, followed by a dot (".") and “begin” or “end”. For example:
 - `begin="otherAnimation.begin"`—This animation starts when “otherAnimation” starts.
 - `end="otherAnimation.begin"`—This animation stops when “otherAnimation” starts.
 - `begin="otherAnimation.end"`—This animation starts when “otherAnimation” ends.

- An event occurs on some element (possibly the element the animation affects), specify the element's ID, followed by a dot and the event name (without the "on" prefix). If an element ID is omitted, the element affected by this animation is used. For example:
 - `begin="startButton.click"`—This animation starts when the SVG element with the ID "startButton" is clicked.
 - `begin="focusin"`—This animation starts when the element affected by this animation receives focus.
- To repeat an animation a certain number of times, use the animation's ID followed by a dot and the function-like **repeat()** declaration. To specify at which iteration to start or stop this animation, add the iteration index inside the parentheses. For example:
 - `begin="loopAnimation.repeat(3)"`—This animation starts on the third time that the "loopAnimation" repeats.
- The user presses a key on the keyboard, use the function-like **accessKey()** declaration. The key is written without quotes. For example:
 - `begin="accessKey(g)"`—This animation starts when the user types the letter "g".

Time can also be specified relative to any of the triggers by appending a plus (+) or minus (-) sign and a clock time. For example, to specify that an animation begin five seconds after the "startButton" element is clicked, the attribute could be written as: `begin="startButton.click+5s"`.

By default, the **begin** attribute is relative to when the SVG document loads, which is considered a kind of trigger. So if the attribute is omitted, the animation begins immediately when the document loads. If only a time is specified for the attribute, the time is relative to when the document loads. For example, `begin="30"` specifies that the animation starts thirty seconds after the SVG document loads.

Special Times

- Real world date and time can be specified by using the **wallclock()** function-like declaration. Inside the parentheses, the programmer can specify any standard ISO8601 date or time format. A simple hour of the day can be specified, or a full-fledged date and time with a time zone offset. This can be useful for running an animation the same time every day, or only on a specific day of the year. For example:
 - `begin="wallclock(9:00:00)"`—9:00 A.M. (every morning)
 - `begin="wallclock(2020-03-15T13:30:00-08:00)"`— 9:30 P.M. UTC on March 15, 2020
- To determine that an animation starts when it is triggered by JavaScript or some other mechanism, the "indefinite" string is used. For example:
 - `begin="indefinite"`—The animation will only start if JavaScript starts it.

The default value for the **end** attribute is "indefinite", so unless a value is especially specified or JavaScript is used to stop the animation, an animation always continues indefinitely.

Multiple Times

The **begin** and **end** attributes can specify a list of times that start or stop an animation. This is useful for animations having more than one criteria for starting or stopping. To specify a list of times, separate them with a semicolon (;). Whenever any condition is met, the animation starts or stops. For example, `end="stopButton.click+3s;synchronizedAnimation.end;begin+1min"` specifies that this animation ends three seconds after the "stopButton" element is clicked, when the "synchronizedAnimation" animation ends, or one minute after this animation begins. Because an animation can be started and stopped

multiple times, every condition can have an opportunity to be applied, especially if the animation is restarted right after it stops.

Duration

The animation elements also support a *duration* attribute called **dur** that specifies how long an animation lasts. The duration attribute can be specified as a clock time or "indefinite". The duration only takes effect after the animations starts.



Note: The **end** and **dur** attributes have some overlap. They are sometimes interchangeable if the value is adjusted. For example, if an animation is triggered by a mouse click and runs for thirty seconds, the **dur** attribute could be specified as `dur="30"` and the **end** attribute could be specified as `end="click+30"`. If the **end** attribute omits the "click+" string, its trigger becomes the document load time, and not the click event. In contrast, because the **dur** attribute specifies how long an animation runs, it always implicitly starts when the animation starts. Using the **dur** attribute in this way can help avoid confusion and potential bugs. However, sometimes the overlap is beneficial, so both attributes are used together. For example, when used together, they can specify that an animation runs for thirty seconds, unless the element is clicked: `dur="30" end="click"`.

The **min** and **max** attributes also affect the duration. The **min** attribute specifies the minimum amount of time that the animation can last, and the **max** attribute specifies the maximum amount of time the animation can last. By default, **min** is 0 and **max** is indefinite, so an animation's duration is not limited in any way. They both accept a clock time, but only the **max** attribute accepts the value "indefinite".

The Animation Elements

To apply an animation to a shape, the animation element is added as a child of the shape's element.

The set Element

The **set** element is the simplest SVG animation element. The **set** element takes one of its parent element's attributes or CSS properties, and then changes its value as specified. In its most basic form, the **set** element has two attributes:

- **attributeName**—Specifies the name of the parent element's attribute to modify.
- **to**—Specifies the new value to apply to the specified attribute.



Note: According to the standard, only the **attributeName** attribute is mandatory. However, if the **to** attribute is omitted, the **set element** has no effect because it does not actually change anything.

The following example changes the parent rectangle's fill color from gray to orange.

Using the set Element

```
<rect x="20" y="20" width="100" height="30" stroke="black" fill="gray">
  <set attributeName="fill" to="orange" />
</rect>
```

The example can hardly be called animation, because the change is not evident. When time is not specified for the **set** element, the change takes effect immediately and lasts indefinitely because of the

begin, **end**, and **dur** attributes' default values previously described. If a finite duration is specified, or if the animation is stopped some other way, the target attribute changes back to its original value after the **set** element's animation stops.

The following example changes the animated element's position temporarily.

Using the **set** Element to Change Position Temporarily

```
<circle r="30" cx="50" cy="50" stroke="black" fill="darkgreen">
  <set attributeName="cx" to="150" begin="click" dur="5" />
</circle>
```

This example draws a circle at (50, 50) and moves it 100 pixels to the right when the user clicks it. The circle stays there for 5 seconds and then automatically moves back. Because the **set** element does not create a transition, the circle appears at (50, 50), then immediately appears at (150, 50) when it is clicked, and then immediately moves back when the duration period is completed.

The **animate** Element

The **animate** element is similar to the **set** element, except that it makes the change gradually, instead of all at once. The **animate** element takes a single attribute or CSS property and progressively changes it.

The following example changes the animated element's position gradually.

Using the **animate** Element

```
<circle r="30" cx="50" cy="50" stroke="black" fill="darkgreen">
  <animate attributeName="cx" to="150" begin="click" dur="5" />
</circle>
```

This example is identical to the previous example, except that an **animate** element is used, instead of a **set** element. Over the animation's five seconds, the browser moves the circle 100 pixels to the right. As soon as the animation ends, the circle moves back to its original position (with no transition).



Note: The browser is responsible for calculating how quickly to refresh the drawing as the shape moves. The faster it refreshes the drawing, the smoother the transition appears, but more refreshes also consume more resources. For example, the browser may refresh the shape every two seconds as it progresses from left to right, but that would create a very jittery animation. Alternatively, it may refresh the drawing every two milliseconds, which might cause the browser to monopolize the CPU. The browser tries to strike a balance between creating a smooth animation and conserving system resources.

In the previous examples, when the animation ends, the modified attribute always reverts to its original value. To specify that the effects of the animation remain after the animation stops, use the animation element's **fill** attribute. This attribute is very different than the **fill** attribute used with a shape element; it does not specify a color or pattern. The animation element's fill attribute can have one of two values:

- "remove"—Indicates that the animation effects are removed and the target attribute restored to its original value.
- "freeze"—Indicates that the animation effects remain unless the animation is restarted.

The default value is "remove".

The following example "freezes" the animation's effects after the animation ends.

Freezing an Animation

```
<circle r="30" cx="50" cy="50" stroke="black" fill="darkgreen">
  <animate attributeName="cx" to="150" begin="click" dur="5" fill="freeze" />
</circle>
```

In this example, the circle is gradually moved 100 pixels to the right, and stays there when the animation ends. However, if the user clicks the element again after the animation ends, the circle is immediately moved back to its original starting point at (50, 50) before the animation starts again.

The **animate** element can change more than just coordinates. The **animate** element can change any value having some numeric representation, including angles and colors.

The following example changes the circle's color by using the animate element.

Animating a Color Change

```
<circle r="30" cx="50" cy="50" stroke="black" fill="gray">
  <animate attributeName="fill" to="orange" begin="click" dur="5" />
</circle>
```

This example works because every color, including every named color, has an underlying numeric representation that the **animate** element can modify gradually.



Best Practice: The SVG standard also has a separate **animateColor** element to animate color changes, such as the one in the example. However, SVG has added support for color animation to the **animate** element. The **animateColor** element is therefore being deprecated and should not be used.

The animateMotion Element

Moving shapes around is a common animation requirement. While the **animate** element can move an element by changing its coordinates, its path of motion is limited and only one coordinate can be specified by a single **animate** element. The **animateMotion** element is more flexible and provides a few options for moving a shape.

The first option is to use the **to** attribute to specify the shape's final position target coordinates. The browser moves the shape along a straight line from its initial position to the final position.

The following example uses the **animateMotion** element to move an element along a simple straight line.

Using the animateMotion Element with a Simple Linear Animation

```
<rect x="50" y="50" width="100" height="40" stroke="black" fill="darkgreen">
  <animateMotion to="150,50" begin="click" dur="3" />
</rect>
```

This example replaces the **animate** element with an **animateMotion** element and specifies target coordinates on the **to** attribute. This causes the browser to move the shape along a linear path to the point at (150, 50).



Note: The SVG standard does not specify which exact coordinates are moved to the target coordinates and leaves it up to the browser to decide. The best approach is to use trial and error during development to ensure the correct motion is acquired, and to test on multiple browsers.

The second option is to move a shape along multiple lines in succession, using the **values** attribute. Its value is a list of coordinates separated by semicolons. The browser moves the shape along a linear path from one point in the list to the next.

The following example uses the `animateMotion` element to move an element along a zigzag path.

Using the `animateMotion` Element with the **values** Attribute

```
<rect x="50" y="50" width="100" height="40" stroke="black" fill="darkgreen">
  <animateMotion values="50,0;100,100;150,0;200,100" begin="click" dur="5" />
</rect>
```

The browser moves the rectangle in the example to (50, 0), then to (100, 100), then to (150, 0) and finally to (200, 100), all within a five second period.

Although this example produces a more complex path for motion, it is still very limited, because the path consists only of straight lines.

Another option is to use the **path** attribute to specify motion along a complex path. The **path** attribute of the `animateMotion` element only supports some of the path commands. Specifically, it supports only the "move to" (M), "line to" (L), "horizontal line to" (H), "vertical line to" (V), "close path" (Z) and "cubic Bezier curve to" (C) commands, and their relative equivalents ("m", "l", "h", "v", "z" and "c").

The following example uses the `animateMotion` element to move an element along a complex path.

Using the `animateMotion` Element with the **path** Attribute

```
<rect x="50" y="50" width="100" height="40" stroke="black" fill="darkgreen">
  <animateMotion path="M 20 50 C 80 0 110 100 200 50 L 110 200 Z" begin="click" dur="5" />
</rect>
```

In this example, the **path** attribute specifies a complex non-linear path that the shape follows for five seconds.

The `animateTransform` Element

The **set**, **animate**, and **animateMotion** elements perform very specific animation tasks. However, often, a more flexible approach is required. The **animateTransform** element specifies general transformations that can be applied to a shape to translate, rotate, scale, or skew it.

The **animateTransform** element is always used to modify the affected element's **transform** attribute, so **attributeName** is always set to "transform".

The **type** attribute specifies the kind of transformation and can have one of the following values:

- "translate"—animates moving a shape
- "scale"—animates resizing a shape
- "rotate"—animates rotating a shape
- "skewX"—animates changing the shape's angles along the x-axis
- "skewY"—animates changing the shape's angles along the y-axis

The **to** attribute specifies the selected transformation's target value. The value specified for the **to** attribute depends on the **type** attribute value. Because each type maps to a specific transformation with the same name, the **to** attribute has the same arguments as the selected transformation. The square brackets indicate optional arguments.

- **translate: *tx* [*ty*]**—Moves the shapes ***tx*** distance horizontally and ***ty*** distance vertically.
- **scale: *sx* [*sy*]**—Resizes the shape by a multiple of ***sx*** horizontally and ***sy*** vertically.
- **rotate: *angle* [*cx cy*]**—Rotates the shape ***angle*** degrees about the point (***cx***, ***cy***) if specified, or the current point.
- **skewX: *angle***—Skews the shape by ***angle*** degrees relative to the x-axis.
- **skewY: *angle***—Skews the shape by ***angle*** degrees relative to the y-axis.

The following example applies several transformations to a shape by using the `animateTransform` element.

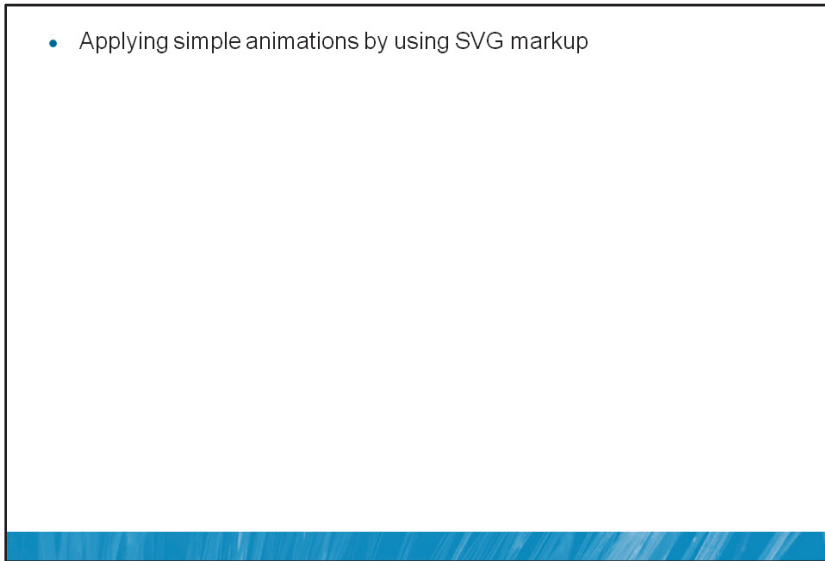
Using the `animateTransform` Element

```
<rect x="50" y="50" width="100" height="40" stroke="black" fill="darkgreen">
  <animateTransform attributeName="transform" type="scale" to="2" begin="click" dur="3"
    fill="freeze" additive="sum" />
  <animateTransform attributeName="transform" type="rotate" to="30" begin="click"
    dur="3" fill="freeze" additive="sum" />
  <animateTransform attributeName="transform" type="translate" to="250, 50" begin="click"
    dur="3" fill="freeze" additive="sum" />
</rect>
```

The example applies three different transformations. The shape is scaled to twice its size in the first transformation. It is rotated thirty degrees by the second transformation. Then, it is moved (translated) to the point at (250, 50). All three transformations are applied at the same time.

By default, when multiple transformations are applied, they override each other, so only the last one takes effect. To allow multiple transformations to work together, the **additive** attribute is set to "sum", instead of its default value, "replace".

Demonstration: Drawing SVG Animations by Using Markup



This demonstration shows how to:

- Animate an SVG element by modifying various attributes over time.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file, add an **svg** element with a blue circle containing various animations, inside the **body** element.

```
<svg version="1.1" xmlns="http://www.w3.org/2000/svg">
  <circle cx="50" cy="50" r="40" fill="blue">
    <animate attributeName="fill" to="yellow" begin="click" dur="5" fill="freeze" />
    <animate attributeName="cx" to="200" begin="click" dur="5" fill="freeze" />
    <animate attributeName="cy" to="150" begin="click" dur="5" fill="freeze" />
    <animate attributeName="r" to="80" begin="click" dur="5" fill="freeze" />
  </circle>
</svg>
```

4. Save the HTML file.
5. Run the saved file.
6. Click the blue circle and observe the animations.

SVG Animation by Using JavaScript

- SVG animations can be added, removed, or modified by using JavaScript
- To create an SVG element, use the **document.createElementNS** method
- SVG animation elements do not expose convenience properties; use **getAttribute** and **setAttribute**
- Time loops can be used to trigger SVG animations

```
var animation = document.getElementById("theAnimation");
animation.setAttribute("attributeName", "fill"); // use the
animation to change the element's fill color
animation.setAttribute("to", "yellow"); // change the fill color
to yellow
animation.setAttribute("dur", "3s"); // run the animation for
three seconds
```

Overview

While declarative SVG animations are powerful, they are often combined with JavaScript. JavaScript serves two purposes when used with SVG animations. It introduces application logic, such as retrieving graph data or updating a game's state in response to user interaction; and it provides finer control over the animations. Sometimes, declarative animations can appear complicated, so JavaScript helps with simplification.

In the animations demonstrated so far, the browser manages the clock while the web developer specifies intervals and durations. For example, the **dur** attribute of any animation element specifies how long the animation lasts, while the browser decides how often to refresh the drawing. For many animations, this is sufficient. However, there are also animations such as game animations that require finer control over the clock.

It is common to use a mixed approach when programming SVG. The first clock is the browser's built-in clock. The SVG document contains animations, or JavaScript code constructs declarative animations and adds them to the document, effectively handing them off to the browser. The second clock is a time loop, like the one used to draw video on the canvas in Module 5, "Getting Started with Graphics and Multimedia Elements", in course 10953. The time loop is used to manage application events, respond to user feedback, and perform application logic.

The SVG DOM

SVG has a DOM. Like HTML's DOM, SVG can be accessed with JavaScript and manipulated in the same way. To control an animation by using JavaScript, the animation element must be retrieved. Any of the standard element retrieval methods work.

The following example shows how to retrieve an animation element.

Retrieving an Animation Element Using `getElementById`

```
var animation = document.getElementById("theAnimation");
```

Starting and Stopping Animations

Every animation element has a few methods that can be used to start or stop the animation. The **beginElement** and **endElement** methods start and stop animations. The methods have the same signature.

```
animationElement.beginElement()
```

```
animationElement.endElement()
```

Neither method returns any value. When the **beginElement** method is called, it does not directly start the animation. The current time is first added to the animation's list of begin values and then the browser triggers the animation. The same is true for the **endElement** method.

There are also two similar methods—**beginElementAt** and **endElementAt**—that can start or stop animations with an offset time.

```
animationElement.beginElementAt(offset)
```

```
animationElement.endElementAt(offset)
```

The **offset** argument represents the number of seconds to add to the current time, before starting or stopping the animation.

The following example starts and stops an animation.

Starting and Stopping Animations

```
var animation = document.getElementById("theAnimation");
animation.beginElement(); // start the animation now.
animation.endElementAt(30); // stop the animation in thirty seconds
```

This code retrieves the animation element with the ID "theAnimation", starts the animation immediately, and instructs the browser to stop the animation in thirty seconds.



Best Practice: Animations omitting the **begin** and **end** attributes start when the document loads, and the animations run indefinitely. However, animations accessed by using JavaScript are usually started and stopped in code. To prevent the animations from starting or stopping on their own, set the **begin** and **end** attributes to "indefinite".

Very often, multiple animations are synchronized with each other. For example, one animation could be triggered by using a mouse click, by setting **begin**="click", and another animation could start when the first animation starts by setting **begin**="firstAnimation.begin". To control these animations by using JavaScript, it is usually enough to start or stop just the first animation that triggers all the rest. In this case, if the first animation is triggered by code, instead of a mouse click, the **begin** attribute is set to "indefinite", instead of "click". The rest of the animations in the chain, although ultimately controlled by JavaScript, are not modified.

Modifying Animations by Using JavaScript

After an animation element is retrieved by using JavaScript, it can be modified. However, the SVG DOM does not expose most attributes as properties. For example, the JavaScript object has no "begin" or "end" property to correspond to the **begin** and **end** attributes. To retrieve attribute values and modify them, use the standard **getAttribute** and **setAttribute** methods.

The following example changes the animation attributes at run time.

Setting Animation Attributes Using JavaScript

```
var animation = document.getElementById("theAnimation");
animation.setAttribute("attributeName", "fill"); // use the animation to change the
element's fill color
animation.setAttribute("to", "yellow"); // change the fill color to yellow
animation.setAttribute("dur", "3s"); // run the animation for three seconds
```

Modifying the DOM

It is also possible to add and remove animations from the DOM by using the standard DOM manipulation methods, including **appendChild**, **removeChild**, and so on. The only real difference between what has already been discussed is that the elements are created by using the **createElementNS** method, which can create elements conforming to other namespaces. In this case, an animation element must be created with SVG's namespace.

The following example creates an **animate** element, adds it to the DOM, and starts it.

Adding Animation Elements to the DOM

```
var animation = document.createElementNS("http://www.w3.org/2000/svg", "animate");
animation.setAttribute("attributeName", "fill");
animation.setAttribute("to", "yellow");
animation.setAttribute("dur", "3s");
animation.setAttribute("begin", "indefinite");
var target = document.getElementById("target");
target.appendChild(animation);
animation.beginElement();
```

This code creates an **animate** element by using the HTML document's **createElementNS** method, and passes it the SVG namespace URL. After it is created, it is modified, added as a child of the element the animation animates, and then started.

Using a Time Loop

Using a time loop to control SVG animations is very similar to using a time loop to draw a video on the canvas.

The first step is to set up the time loop in JavaScript.

A Basic Time Loop

```
$(function() {
  function frame() {
    // do something here
    window.setTimeout(frame, 100); // trigger the next iteration
  }
  frame(); // trigger the first iteration
});
```

This example is very similar to an example in Module 5. This example uses jQuery to initialize the time loop when the document loads, by embedding an anonymous function inside the **\$** function. It then creates a method called **frame** that runs application logic and then instructs the browser to call the **frame** method again after a certain interval has passed, in this case, 100 milliseconds.

Now all that is left is to perform application logic.

The following example implements a very simple countdown timer.

SVG Countdown Timer

```
$(function() {
  var time = 600; // count down from 10 minutes (= 600 seconds)
  var clock = document.getElementById("clock");
  var animation = document.getElementById("highlight");
  function frame() {
    time--; // decrement the second
    var minutes = Math.floor(time / 60); // calculate the number of whole minutes left
    var seconds = Math.floor(time % 60); // calculate the number of whole seconds left
    clock.textContent = (minutes < 10 ? "0" : "") + minutes + ":" + (seconds < 10 ? "0" :
    "") + seconds;
    if(time % 10 === 0) {
      // trigger the "highlight" animation every 10 seconds
      animation.beginElement();
    }
    if(time > 0) {
      // trigger the next time loop iteration if the timer has not yet reached 0
      window.setTimeout(frame, 1000);
    }
  }
  frame(); // start the countdown
});
```

This code demonstrates the use of a time loop to trigger SVG animations. The time variable stores the remaining seconds and is reduced by one second every time the frame function runs. The expression, `time % 10`, returns zero whenever the number of seconds remaining is evenly divisible by 10. So every ten seconds, the **beginElement** method is called on the **animation** element, triggering the animation.

The following example shows the corresponding HTML document with embedded SVG.

An Embedded SVG Countdown Timer

```
<body>
  <svg xmlns="http://www.w3.org/2000/svg">
    <rect x="50" y="50" width="110" height="40" stroke-width="3" stroke="black"
    fill="whitesmoke">
      <animate id="highlight" dur="200ms" attributeName="fill" to="orange" fill="freeze"
      begin="indefinite" />
      <animate dur="1" attributeName="fill" to="whitesmoke" fill="freeze"
      begin="highlight.end" />
    </rect>
    <text x="105" y="74" id="clock" alignment-baseline="middle" text-anchor="middle"
    font-family="Arial" font-size="42px"></text>
  </svg>
</body>
```

This HTML document contains an embedded SVG document. The SVG draws a rectangle and some text. The previously described JavaScript code modifies the text by changing the **clock** variable's **textContent** property. When the **beginElement** method is called on the **highlight** animation element, the rectangle's **fill** is changed to orange over 200 milliseconds. The second animate attribute is tied to the first one. Its **begin** attribute is "highlight.end", so it is triggered every time the **highlight** animation finishes, and it takes one second to restore the whitesmoke color to the parent rectangle.

This example demonstrates the use of application logic (in this case, the timer countdown logic) in a time loop to trigger declared animations. This technique can be used and can support very advanced animated graphics applications by using SVG and JavaScript together.

Canvas Animation by Using JavaScript

- The canvas can only be animated by using JavaScript
- A time loop is used to control the animations and the application logic
- The canvas surface has to be cleared and redrawn completely for every iteration

```
$(function () {
  function frame() {
    // do something here
    window.setTimeout(frame, 100); // trigger the next iteration
  }

  frame(); // trigger the first iteration
});
```

The canvas has no equivalent to SVG's declarative markup. Animations can only be created by using JavaScript code. Some animations created with SVG are not very difficult to reproduce with the canvas. However, creating smooth transitions and effects like SVG requires skill and experience.

As a quick reminder, the canvas uses immediate-mode graphics and does not remember much beyond the bitmap array of pixels drawn on its surface and its own state. It has no innate data structure for representing the drawn objects. The web developer must create those data structures and keep track of their state in JavaScript.

This topic illustrates how to use some of the same techniques that have been introduced earlier to create basic animations with the canvas.

Using the Time Loop

Just as the time loop was used in Module 5 to draw a video onto the canvas and in the previous topic to manage the application logic, it can be used with the canvas to manage any kind of animation. Because the canvas has no built-in mechanism to manage animations and has no internal clock of its own, multiple time loops are often used to manage different animations. There is usually a master time loop, often called a "game loop" or "application loop", which manages the main application logic and spins off smaller loops to draw animations.

The following examples construct a countdown timer similar to the one built by using SVG.

The following example starts with a basic time loop.

Another Basic Time Loop

```
$(function() {
  function frame() {
    // do something here
    window.setTimeout(frame, 100); // trigger the next iteration
  }
  frame(); // trigger the first iteration
});
```

Just as in previous examples, jQuery calls the initialization code, creating the infinitely-repeating time loop.

The application logic to calculate the time, construct the remaining time display string, and end the loop is also the same.

The following example illustrates the basic application logic for the countdown timer.

Basic Countdown Timer Application Logic

```
$(function() {
  var time = 600;
  function frame() {
    time--;
    var minutes = Math.floor(time / 60);
    var seconds = Math.floor(time % 60);
    var text = (minutes < 10 ? "0" : "") + minutes + ":" + (seconds < 10 ? "0" : "") +
seconds;
    if(time > 0) {
      window.setTimeout(frame, 1000);
    }
  }
  frame();
});
```

The application logic for this application is simple. However, this code does not actually display anything. To draw on the canvas, a two-dimensional context has to be retrieved first. In the SVG example, the only tasks that JavaScript had to perform were to update the time display and trigger the animation. The SVG DOM managed the rest. In this case, there is no DOM, so the code has to draw the rectangle and the text.

The following code adds the countdown timer's rectangle and the text.

Basic Countdown Timer with the Canvas

```
$(function() {
  var time = 600;
  var context = document.getElementById("canvas").getContext("2d");
  context.font = "42px Arial";
  context.textAlign = "center";
  context.textBaseline = "middle";
  context.lineWidth = 6;
  function frame() {
    time--;
    var minutes = Math.floor(time / 60);
    var seconds = Math.floor(time % 60);
    var text = (minutes < 10 ? "0" : "") + minutes + ":" + (seconds < 10 ? "0" : "") +
seconds;
    context.clearRect(0, 0, context.canvas.width, context.canvas.height);
    context.strokeRect(50, 50, 110, 40);
    context.fillStyle = time % 10 === 0 ? "orange" : "whitesmoke";
    context.fillRect(50, 50, 110, 40);
    context.fillStyle = "black";
    context.fillText(text, 105, 72);
    if(time > 0) {
      window.setTimeout(frame, 1000);
    }
  }
  frame();
});
```

After the context is retrieved, it is initialized.

Every time the loop runs, the canvas is cleared with a call to the context's **clearRect** method. If the **clearRect** method is not called, the canvas draws over what was already drawn. This may not make a difference for the rectangle, because it is drawn exactly the same way in every iteration, but the text cannot be drawn over previously drawn text. It has to be erased first and the **clearRect** method erases everything from the specified rectangle.



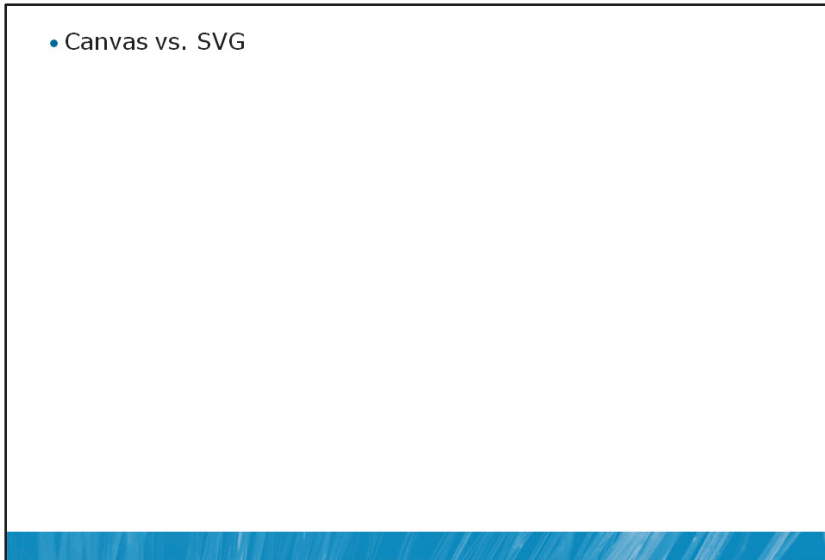
Best Practice: Clearing the drawing surface at the beginning of every iteration is required when working with the canvas, although it can be optimized if only a small area has to be redrawn.

After the canvas is cleared, the outer rectangle and the text are redrawn.

Finally, the SVG animation is able to create a smooth gradual transition from a whitesmoke-colored background to an orange background, and back again. Doing this smoothly with the canvas is a bit more difficult. In the last example, the **fillStyle** is changed to orange every ten seconds, and changed back to whitesmoke at the next iteration.

Lesson 3

Choosing Between SVG and Canvas



HTML5 has two graphics models: SVG and Canvas. Though they are very different from one another, each of them can be used to draw advanced and attractive graphics. They both offer general-purpose graphics solutions suitable for many types of applications, and they are both supported by almost every modern browser. Making sense of the differences and choosing which model to use in a specific application can be difficult. This topic aims to shed some light on the matter and offer guidance as to the types of problems that each of the two models is better suited to solve.

Lesson Objectives

After completing this lesson, you will be able to choose between SVG and the Canvas.

Canvas vs. SVG

Canvas	SVG
Immediate-mode bitmap graphics	Retained-mode vector graphics
Simple low-level API	Rich abstract API
Javascript	Markup and JavaScript
Better for smaller screen or more objects	Better for larger screens or fewer objects
Animation is difficult	Animation is easy
Difficult to support interactivity	Easy to support interactivity
Does not support accessibility	Supports accessibility

Differences

Abstraction

The canvas is a bitmapped drawing surface—basically an array of pixels with colors. Resizing the drawing surface does not cause the canvas to resize its contents, or even to refresh the drawing. If portions of a drawing are scaled, skewed, or otherwise transformed, the quality of the image may be reduced. The canvas is an immediate-mode graphics system, which has the following implications:

- Drawing commands are issued by the programmer and sent directly to the drawing surface. The canvas does not store any information on the series of commands that produced the drawing.
- The API has relatively few methods and properties.
- The API exposes low-level access to the bitmap information, including support for modifying pixel data.
- The web developer has to create and maintain the object model used to create the drawing. This may require more effort, but allows for optimization.

SVG uses scalar graphics and stores a lot of information about the drawing in memory, including shapes, filters, and animations. If the drawing surface is resized or just parts of the drawing are transformed, SVG can resize and refresh the drawing without losing any quality. SVG is a retained-mode graphics system, which has the following implications:

- The programmer does not issue commands directly to the drawing surface. Rather, the programmer modifies objects, and the browser converts these modifications into commands that it sends to the drawing surface.
- The API is very rich.
- The API does not expose low-level pixel information.
- SVG maintains the object model used to create the drawing; the web developer merely updates the objects. It is easier, but less flexible.

Code

Canvas uses a single HTML element, and is manipulated entirely by JavaScript code. Although some parts of the Canvas API are based somewhat on CSS, CSS style sheets do not directly interact with or affect the canvas.

SVG, meanwhile, uses many graphical elements representing different aspects of the drawing, such as shapes or filters, and is embedded in an HTML document. Its elements become part of the DOM and can be accessed and modified by using JavaScript. In addition, CSS can be applied to SVG elements.

Event Model

When an event is raised by a canvas, the web developer has to write low-level code to interpret it. For example, if a user clicks a specific part of the canvas, the developer has to perform hit testing to determine which objects were drawn at the specific x and y coordinates of the event.

SVG automatically maps events to shapes, so programmers respond to high-level events that are already mapped to the objects that they apply to. For example, if a user clicks a shape, that shape raises an event, and the programmer can more easily determine how to respond to the event.

Performance

The canvas uses only as much memory as it needs to represent the bitmap and the canvas's state, so it generally consumes less memory. Because the canvas does not store information about the object model, the number of objects drawn on the canvas is negligible, although the number of objects may affect the amount of memory consumed by the programmer's code. However, a larger canvas stores more data and takes longer to render. Overall, the canvas provides better performance when the drawing surface is smaller or when a graphics application has to manage a large number of objects.

SVG stores a lot of information about each of its drawing objects, so it tends to use more memory, especially as the number of objects grows. On the other hand, because it uses vector graphics, the size of the canvas has little effect on performance. Overall, SVG provides better performance when the number of objects is smaller or the surface is larger.

Accessibility

At the time of this writing, the canvas has almost no support for accessibility, although W3C is considering the issue. Because the canvas surface is purely composed of pixel data and has no markup, accessibility issues are very difficult to solve.

SVG has built-in support for accessibility. Not everything in a graphics application can be made accessible, but SVG is designed with some accessibility in mind.

Similarities

SVG and Canvas have more differences than similarities, but they do share some things in common. In particular, they both try to make use of existing standards and conventions, so knowledge in one area can translate into knowledge in the other.

- They are both standardized by W3C.
- Both are general-purpose graphics solutions.
- Both perform reasonably well in most applications.
- Both can be modified by using JavaScript.
- Both rely on CSS. Although the canvas is not directly affected by CSS style sheets, its specification uses CSS syntax and values for colors, fonts, and other style information.

- Both use similar—but not identical—shape primitives in their paths. For example, they both have "move to", "line to", and "cubic Bezier curve to" path commands.
- They use similar language where possible. For example, they both use the terms "stroke" and "fill" in the same way.
- Increasingly, browsers use hardware acceleration to render both models.

Choosing SVG or Canvas

Comparing the pros and cons of SVG and Canvas can help pinpoint the solution most appropriate for the type of graphics required by a given application.

Canvas is usually better suited for the applications that:

- Require low-level access to pixel data, such as applications that make use of video manipulation or screen captures
- Need small drawing surfaces and require little or no interaction with the user, such as those used for web advertising.
- Have many objects or complex scenes and need better overall performance.

SVG is usually better suited for displaying:

- Static images.
- High-fidelity documents for viewing or printing.
- Applications that rely on vector graphics support, such as drawing applications.
- Applications that require accessibility.

SVG and the Canvas are not suitable for every purpose. Applications that support text entry and other features that browsers already provide should avoid re-implementing these features, by using SVG or the Canvas. Browsers have many accessibility features that are very difficult and time-consuming to re-implement. Programmers should avoid using SVG or the Canvas when building applications that require:

- Forms, such as logon or registration forms. Web developers should use standard HTML forms.
- Text editing applications, such as word processors and spreadsheets. Web developers can use HTML and advanced JavaScript techniques to create complex applications without SVG or Canvas.

Some types of applications should choose their graphics models on a case-by-case basis. There may not be a clear, general consensus on which type of graphics medium is better suited. These applications include:

- Interactive charts and graphs
- 2D games



Note: Note that these are merely guidelines and not strict rules. Other, non-technical considerations may also affect the choice between SVG and Canvas. For example, a strong personal preference, or greater familiarity with one or the other, can easily tip the scales.

Lab: Creating Advanced Graphics

- Adding an SVG Chart with Animation
- Adding Animation to the Game Details Canvas-Based Slideshow

Logon Information

Virtual Machine	Virtual machine: SEA-DEV
User Name	User name: Admin
Password	Password: Pa\$\$w0rd

Estimated Time: 60 minutes

Scenario

The Contoso management has decided that they want additional features on the Game Details webpage. The new features that they want are:

- Game-buying statistical data. The data will show the game's number of sales for each month, over the course of the previous year.
- Animation on the slideshow.

Objectives

After completing this lab, you will be able to:

- Create an animated chart by using SVG.
- Add animation to the canvas-based slideshow.

Exercise 1: Adding an SVG Chart with Animation

Scenario

You have been asked to add an SVG bar chart to the Game Details webpage to demonstrate its feasibility. The chart should have a bar for each quarter. When the page loads, the bars should “grow” until they reach their actual value, to draw attention to the chart.

For the purposes of demonstrating this chart, you are provided fictitious quarterly sales numbers.

► Task 1: Add an SVG chart to the Game Details webpage.

1. Open the Starter solution.
2. Open the game details view.
3. Add a title to the chart by using SVG.
4. Draw the chart’s axes.

► Task 2: Add data and animation to the bar chart.

1. Add data bars to the bar chart.
2. Add animation to all the bars to start growing shortly after the page loads.

► Task 3: Add CSS styles to the SVG chart.

1. Set the chart’s basic styles.
2. Set the styles for the **text** elements.
3. Set the styles for the bar chart’s bars.
4. Test the application.

Results: Game statistics are integrated into the Game Details page.

Exercise 2: Adding Animation to the Game Details Canvas-Based Slideshow

Scenario

The management likes the new canvas-based slideshow, but feels that the slide changes are not emphasized enough. You are instructed to add an animated transition every time the slide changes.

► Task 1: Create an isolated time loop for the transition's animation.

1. Open the JavaScript file.
2. Create a **transition** object to store information about the animation.
3. Create the basic time loop and a special function to trigger it.
4. Constrain the animation to a specific number of animation frames.
5. Start the transition's animation when the main slide changes.

► Task 2: Initialize the animation.

1. Set the transition's animation to start at the current thumbnail's location and end at the main slide's location.
2. Initialize the current step every time the transition is started.

► Task 3: Draw each frame of the animation.

1. Get a reference to the canvas' 2D context.
2. Constrain the drawing surface for the animation to the area occupied by the main slide. Do not draw on the thumbnail area.
3. Draw the current slide's image at the intermediate location for the current frame of the animation.

► Task 4: Prevent the main time loop from drawing, while the animation is in progress.

1. Add a Boolean flag to determine whether the main time loop can draw or an animation is in progress.
2. Toggle the Boolean flag when the animation starts and ends.
3. Do not draw the main slide when the animation is in progress.
4. Test the application.

Results: Animation is integrated into the Game Details page.

Module Review and Takeaways

This module introduced the second graphics model included in HTML5: SVG. It also compared SVG and the Canvas, and showed how to add animation to SVG and to the Canvas.

Module 7

Using Client-Side Storage

Contents:

Module Overview	7-2
Lesson 1: Web Storage vs. Cookies	7-3
Lesson 2: Web Storage API	7-9
Lab: Using Client-Side Storage	7-18
Module Review and Takeaways	7-21

Module Overview

- Web Storage vs. Cookies
- Web Storage API

Data persistence is one of the most important issues while developing sites and applications. Most sites and applications today have a persistence mechanism, which allows them to store valuable data that will be presented to the users. This storage can be a database, or files in the file system (with various formats such as XML, text, or Excel. Even so, when developing an application, there are times when you want your data readily accessible to the site and application, to increase performance and minimize network latency. This is where the new client-side storage capabilities in HTML5 can help.

HTML5 introduces new client-side storage Application Public Interface (API), which are implemented natively in web browsers. These mechanisms change the way developers create websites and applications. In the past, only smart-client applications, which are mainly desktop applications, could use local storage on the client computer. Now, websites and applications can also have such functionality by using APIs such as web storage and IndexedDB.

In this module, you will learn about web storage and how to use its API to have a persistence mechanism on the client-side. You will also be introduced to IndexedDB at a very high level because this specification is currently in development and might change in the future.

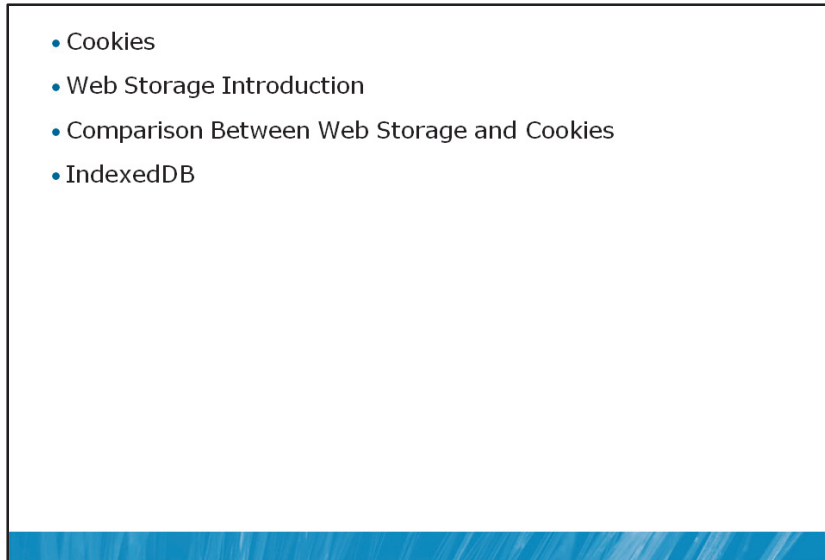
Objectives

After completing this module, you will be able to:

- Use the web storage API.
- Differentiate between session storage and local storage.

Lesson 1

Web Storage vs. Cookies



Before HTML5, websites and application developers had very few options for maintaining data on the client side. To save data for further client-side use, developers could use cookies or specific browser APIs such as the `userData` in previous versions of Internet Explorer. This situation made creating offline websites and applications very difficult, forcing the data persistence burden onto the server-side and increasing network latency because every piece of data needed to be retrieved from the server.

HTML5 presents new APIs that help to store data on client-side. Some of these APIs can be found in most browsers today and some APIs are still in development. In this lesson, you will learn first about client-side storage options before HTML5, and then you will be introduced to web storage, which is one of the mechanisms for data persistence introduced in HTML5.

Lesson Objectives

After completing this lesson, you will be able to:

- Describe the rationale behind using web storage.
- Describe cookies.
- Describe web storage.
- Describe the difference between web storage and cookies.

Cookies

Cookies:

- Simple model for storing and transferring data in client-side
- Include very limited amount of space (mostly 4 KB)
- Transferred back to the server in every HTTP request

```
var expiredate=new Date();
expiredate.setDate(expiredate.getDate() + 20);
var cookieData="data" + "; expires="+ expiredate.toUTCString();
document.cookie= "cookieName=" +cookieData;
```



Cookies are an old and simple persistence mechanism that developers use very widely in websites and applications. A cookie is a small piece of string data up to 4 kilobytes (KB) that is saved in the browser memory or as a text file in the file system on the client. The cookie is transferred in each HTTP request and HTTP response from the client to the server and back. Cookies are mainly used to identify a user, to store state information, user preferences, and so on.

How Cookies Work

When a client requests a webpage from the server, the first request does not contain a cookie. The server identifies that the client has no cookie and it generates one with some data. The server sends the cookie to the client, and from then on, the client sends the cookie in every request and the server sends the cookie in every response.

Every cookie has an expiration date and there is no default value for the expiration date. You can set that date on the server or on the client to be any future date or even set not expire, that is, infinite. The browser stores the cookie until the specified date and time is reached or until the user cleans the browser cookies. If you set the expiration date of a cookie to a date earlier than the current date and time, the cookie expires as soon as the browser receives it. If you do not indicate any expiration date, the cookie is created but it is not stored on the client side.

Cookies Limitations

There are several limitations that you should be aware of when deciding how to use cookies.

- Most browsers limit cookie size to 4 KB. This, of course, enables the developer to save only a small amount of data in the cookie.
- Browsers impose limitations on the number of cookies a website or application can have. Most browsers allow only 20 cookies per site or application.
- Users can disable cookies. Therefore, depending on cookies might result in reduced or limited functionality.

- The data in cookies can be intercepted while it is being sent from the server to the client and from the client to the server.

All these limitations make the cookie mechanism very specific for issues such as identifying users, and less appropriate for data-driven sites and applications.

Using Cookies in JavaScript

JavaScript enables the creation and retrieval of cookies. You will use the `document.cookie` variable to do that.

The following is an example of setting a cookie.

Setting a Cookie

```
var expiredate=new Date();
expiredate.setDate(expiredate.getDate() + 20);
var cookieData="data" + "; expires="+ expiredate.toUTCString();
document.cookie= "cookieName=" +cookieData;
```

In the preceding example, an expiration date is set in the `expireDate` variable. Then, cookie data is created with its expiration date. The data is separated from the expiration date by using a semicolon. Finally, the `document.cookie` property is set with the cookie name and data.

As you can see in the example, creating a cookie is simple and involves setting the `document.cookie` property with a string that includes the cookie name, the equal sign, and the cookie data.

The following is an example of retrieving cookie data.

Retrieving Cookie Data

```
var cookieInfo = document.cookie;
```

Web Storage Introduction

Web Storage:

- Native browser-persistent data storage
- Represented with a dictionary object
- Greater storage capacity and better programmatic interfaces than cookies

Web storage is a new HTML5 JavaScript API specification that specifies methods and protocols used for storing data in a browser. It supports persistent data storage, similar to cookies, and window-local storage, which is located client-side. It is an improvement over using cookies, providing greater storage capacity and better programmatic interfaces. On the other hand, it is very different from cookies in many ways.

The web storage API introduces two new JavaScript objects: `sessionStorage` and `localStorage`. Both of these objects are implemented as a dictionary, which is available for storing data. The data in web storages is kept as strings, and if you want to use JSON objects, you will have to transfer them into string representation (by using the `JSON.stringify` function). The storages are created per domain, which means that for every domain, a storage object might be created (if there is no web storage usage, no storage object is created). The amount of stored data differs between browsers, but most browsers can store up to 5 megabytes inside a single web storage.

The web storage API includes methods to set and get stored data. Also, developers can create handlers for events that are exposed by the web storage API. This makes the web storage API a very good tool for client-side persistence.

Comparison Between Web Storage and Cookies

	Web Storage	Cookies
Max Size	Measured in MB (per browser implementation)	Measured in KB
Server-side knowledge	No	Sent with every HTTP request
Permissions	Per domain	Per domain and path
Access	By built-in JavaScript object	String manipulation
Event Handlers	Supported	Not supported
When it is deleted	Manually using API or when the browser closed on some occasions	When browser is closed or as per timeout

As already indicated, web storage is an improvement on cookies. The two mechanisms differ in many aspects.

Max size. Web storage is measured in megabytes (mainly 5 MB in most browsers). This makes its capacity larger than the 4 KB that cookies can hold.

Server-side knowledge. Cookies are sent in every HTTP request and response. Because this is standard performance, the server has knowledge about the cookies and their data. On the other hand, web storage is a client-side-only mechanism. This means the data stored in the web storage is known only to the browser and is not transmitted to the server. If you want to update the server with data the web storage holds, you will need to create that functionality.

Permissions. Cookies permissions are per domain and path, while web storage permissions are per domain only. This makes web storage more secure.

Access. Web storage introduces built-in JavaScript objects (`localStorage` and `sessionStorage`), which can be manipulated with built-in methods. Cookies are created as strings and can be manipulated with strings.

Event Handlers. Cookies do not support event handlers. Web storage supports an event handler (`onStorage`) that can be registered.

When it is deleted. Web storage includes an API for deleting stored data or even clearing all the storage. It can also delete all the data when the browser is closed by using `sessionStorage` (more about this in the next lesson). Cookies have timeouts that dictate when the cookie expires.

IndexedDB

- Web storage is not a database
- IndexedDB API is an indexed database specification currently being developed
- Will be used for data-driven scenarios, which web storage cannot support

Web storage is implemented as a dictionary. This makes it very limited and it cannot be treated as a database. For a small amount of data, it might be sufficient, but for large amounts of data, it is very limited. IndexedDB is a new HTML5 specification for client-side storage of significant amounts of data. It enables saving structural data that is not necessarily a string (for example, saving JSON objects) like web storage, and it enables retrieving it by using indexes. Even so, the specifications are currently in development and the API is not supported in most major browsers.

IndexedDB has two separate APIs—one for synchronous access and the other for asynchronous access. This makes IndexedDB very flexible as opposed to web storage, which is synchronous only. To use the IndexedDB APIs, you use the `window.indexedDB` attribute.



Additional Reading: For further information about IndexedDB, go to: [IndexedDB](#).

Lesson 2

Web Storage API

- Web Storage Types
- Web Storage Functions and Attribute
- Web Storage Events
- Security and Privacy Considerations

Web storage introduces two related mechanisms for storing structural data on the client-side: **localStorage** and **sessionStorage**. Both mechanisms are represented as native JavaScript objects and have the same simple API functions and attributes. They also include event handlers.

In this lesson, you will get to know the web storage objects and understand where to use each of them. You will also learn about web storage API functions and attributes, and event handlers. In addition, you will be presented with security and privacy considerations for using web storage.

Lesson Objectives

After completing this lesson, you will be able to:

- Evaluate the options available when implementing the web storage API.
- Integrate web storage into webpages to store data on the client-side.
- Integrate storage events.
- Describe the security and privacy considerations with web storage.

Web Storage Types

- Web storage offers two different storages types:
 - `localStorage`
 - `sessionStorage`
- The storage types differ in scope and lifetime

Web storage specifications include two storage types for storing data client-side. Both types are represented by native JavaScript objects and created as a dictionary that holds key/value pairs. Every pair is identified by its key and holds only one string value. That value can also be the string representation of JSON objects that were serialized into strings by using the `JSON.stringify` function. The difference between the two types is their stored data scope and lifetime.

localStorage

Represented by the **localStorage** object. This type of web storage is designed to store data even when the session between the client and the server ends. This means that closing the browser will not clear the storage and the data will still be available if the user goes back to the same domain. The storage also spans multiple windows and tabs, so if two tabs or windows have a session with the same domain, they will share the same **localStorage** data.

sessionStorage

Represented by the `sessionStorage` object. This type of web storage is designed to store data per-page-per-window and is limited to the lifetime of the window. `sessionStorage` is intended to allow separate instances of the same website or application to run in different windows without interfering with each other. When the window is closed, all data in the `sessionStorage` object is cleared.

Web Storage and Concurrency

Multiple browsing contexts/tabs can access storage areas simultaneously. When the browser tries to manipulate a storage object, it first needs to lock it. Only after the browser obtains a lock on the storage can it make the changes.



Note: The concurrency implementation is specified in the web storage specifications: [Web Storage and Concurrency](#)

Web Storage Functions and Attribute

Web storage:

- Includes simple API functions and one attribute
- Supports the length attribute that indicates the number of stored key/value pairs
- Supports all functions in localStorage and sessionStorage

The web storage API is very simple and works the same way for both storage types. It includes only five functions and one attribute.

length attribute. The length attribute retrieves the number of key/value pairs in the storage.

The following is an example of how to use the attribute with the localStorage object.

length Attribute

```
var storageLength = localStorage.length;
```

clear function. As its name indicates, the clear function will remove all the key/value pairs that exist in the storage.

The following example shows how to use the clear function on the localStorage object.

clear Function

```
localStorage.clear();
```

getItem function. The getItem function gets a key as a parameter and retrieves the value associated with that key.

The following is an example of how to get an item stored in localStorage by using the getItem function.

getItem Function

```
var item = localStorage.getItem("itemKey");
```

In the preceding example, the itemKey is the name of the key associated with a specific value in the storage.

setItem function. The setItem function takes two parameters: the key and the value. If the key exists in the storage, its value will be updated to the given value. If the key does not exist, the function creates a new pair inside the storage with the given key and value.

The following is an example of how to store a pair in the localStorage.

setItem Function

```
localStorage.setItem("key", "value");
```

removeItem. The removeItem function gets a key parameter and removes the key/value pair with the given key. If the key does not exist in the storage, the function does nothing.

The following is an example of how to remove the key/value pair by using "someKey" as the key from the localStorage.

removeItem Function

```
localStorage.removeItem("someKey");
```

key function. The key function gets an index as a parameter and returns the name of the key located in that index. The order of the keys is a browser-specific implementation. If the given index is greater or equal to the length of the storage, the function returns null. You should always remember that the indexes in JavaScript start with 0. This means that retrieving the second key provides 1 as the value of the key function.

The following is an example of retrieving the second key in the localStorage.

key Function

```
var key = localStorage.key(1);
```

Demonstration: Web Storage Functions and Attribute



This demonstration shows how to:

- Create a new Empty Web Application in VS2010
- Add a new HTML file
- Add markup to the web page using the new HTML5 Web Storage functions

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file's body element add the following elements:

```
<body>
  <form id="form1">
    <p>
      You have viewed this page <span id="counter"></span> times.
    </p>
  </form>
</body>
```

4. In the HTML file's head element add the following script tag:

```
<script type="text/javascript">
  function setCounter() {
    if (!localStorage.getItem('numberOfPageLoads')) {
      localStorage.setItem('numberOfPageLoads', 0);
    }
    localStorage.setItem('numberOfPageLoads',
      parseInt(localStorage.getItem('numberOfPageLoads')) + 1);
    document.getElementById('counter').innerHTML =
      localStorage.getItem('numberOfPageLoads');
  }
  window.addEventListener("load", setCounter, false);
</script>
```

5. Save the file as an HTML file
6. In **Solution Explorer**, click on project with the right mouse button.
7. Click on the Set As Startup Page menu item.
8. Run the project by pressing CTRL+F5.

Web Storage Events

- The web storage API exposes the onstorage event
- The event is raised when a change occurs in the storage, such as setting a value
- Can help to create tasks such as validation in the web storage

```
addEventListener("storage", logEventProperties, false);

function logEventProperties(e) {
  console.log(e.key);
  console.log(e.oldValue);
  console.log(e.newValue);
  console.log(e.url);
  console.log(e.storageArea);
}
```

Web storage exposes the **storage** event, which is raised when the following API functions are called: **setItem**, **removeItem**, and **clear**, and only when the new value is not the same as the old value. This enables the developer to create an event handler, which is used for purposes such as validation and applying business rules.

The onstorage Event

The onstorage handler gets as a parameter an event object that contains the following attributes: key, oldValue, newValue, url, and storageArea.

- **key**. The named key that was added, removed, or modified.
- **oldValue**. The previous value, which is now overridden or null if a new item is added to the storage.
- **newValue**. The new value to insert with the associated key or null if an item is removed.
- **url**. The url of the page that triggered the change in the storage.
- **storageArea**. A reference to the web storage element that raised the event.

When the clear function is called, the key, oldValue, and newValue are set to null.

The following is an example of wiring the onstorage event to an event handler.

onstorage Event

```
// The onstorage event wire-up
window.addEventListener("storage", logEventProperties, false);
// the event handler which prints to the console all the event properties
function logEventProperties(e) {
  console.log(e.key);
  console.log(e.oldValue);
  console.log(e.newValue);
  console.log(e.url);
  console.log(e.storageArea);
}
```


Demonstration: Web Storage Events

- Creating a simple webpage using the new HTML5 Web Storage event

This demonstration shows how to:

- Add markup to the web page using the new HTML5 Web Storage event.

Demonstration Steps

1. In the HTML file's head element add the following script statement at the end of the script tag:

```
function logEventProperties(e) {  
    console.log(e.key);  
    console.log(e.oldValue);  
    console.log(e.newValue);  
    console.log(e.url);  
    console.log(e.storageArea);  
}  
window.addEventListener("storage", logEventProperties, false);
```

2. Save the file as an HTML file.
3. In **Solution Explorer**, click on project with the right mouse button.
4. Click on the Set As Startup Project menu item.
5. Run the project by pressing CTRL+F5.

Security and Privacy Considerations

- The data stored in web storage is public
- Should not hold secured data
- Should clear storage areas from time to time



Using client-side storage might raise security and privacy questions by the developers and users. For example, is the storage data secured? What data should not be inside the storage? The data in web storage should be considered public data. Therefore, secured data such as serial numbers or passwords must not be saved in web storage.

The web storage specifications provide instructions to browser vendors about security threats the browser should try to avoid, such as Domain Name System (DNS) spoofing attacks (a host claiming to be in a certain domain but in reality not coming from that domain), and cross-directory attacks (different website or application authors sharing one host name and being able to obtain data related to other authors). The specifications also describe how to avoid privacy problems such as user tracking.

 **Additional Reading:** For additional reading about privacy and security, go to the following specification link: [Web Storage Security and Privacy](#)

Lab: Using Client-Side Storage

- Creating the Cart Behavior by Using localStorage
- Integrating the Cart with the Web Application

Logon Information

Virtual Machine	Virtual machine: SEA-DEV
User Name	User name: Admin
Password	Password: Pa\$\$w0rd

Estimated Time:45 minutes

Scenario

During the process of buying games, a user will have a client-side cart that stores all of the current purchasing data. This data includes a list of games the user has added into his or her cart, and several details about those games. Contoso management asks you to develop this feature.

Objectives

During the lab you will:

- Implement the online store cart by using the localStorage API.
- Integrate the online store cart implementation to the webpages.

Exercise 1: Creating the Cart Behavior by Using localStorage

Scenario

In this exercise, you will use the `localStorage`, and the `Cart` and `Game` objects, to create the activity of storing the user's current cart on client-side. Most of the functionality will be created in the `GlobalScripts.js` file, which is known in all the webpages in the application. You will provide support for adding and removing games to the cart. You will also provide support for getting the current total price of the cart.

► Task 1: Create the cart behavior.

1. Open the starter solution, which is located in the `D:\LabFiles\Starter\M07` folder.
2. Add a check whether **localStorage** is available in the browser by using **Modernizr**.
3. Add the **createEmptyCart** function. The function should clear the `localStorage` and add a new `Cart` object to it with the "cart" key.
4. Create the **getTotal** cart function. The function should calculate and return the total price of all the games located inside the current cart.
5. Create the **addGame** cart function. The function should add a new game object in the current cart's games array.
6. Create the **removeGame** cart function. The function should first check whether the supplied game ID exists in the cart's games array. Then, it should remove it from the array and from the row in the cart table in the UI.
7. Create the **handleNoLocalStorage** function. The function should inform the user that the browser does not support `localStorage`.
8. Create the **updateTotalPrice** function. The function should update the total price of the cart in the UI.

Results: A JavaScript file containing all the relevant functions is created.

Exercise 2: Integrating the Cart with the Web Application

Scenario

After you create the cart's behavior, including adding and removing games, you will integrate this behavior into the application webpages. Most of the work is to wire click events by using jQuery.

► Task 1: Integrate cart behavior on the Cart webpage.

1. Add the functionality to create the cart's content in **CartScripts.js**. You should create a new row for each existing game inside the cart table. Then, add the functionality of creating the cart table in the `contentLoaded` function.

► Task 2: Integrate the cart behavior on the Checkout webpage.

1. Add the functionality to create the cart's content in the **CheckoutScripts.js**. You should create a new row for each existing game inside the cart table. Then, add the functionality of creating the cart table in the `contentLoaded` function.

► Task 3: Integrate the cart behavior on the GameDetails webpage.

1. Add the functionality of the "add to cart event handlers" in the **GameDetailsScripts.js**. The handlers should create a game object and insert it in the current cart.

► Task 4: Integrate the cart behavior on the Home webpage.

1. Add the functionality of the "add to cart event handlers" in the **HomepageScripts.js**. The handlers should create a game object and insert it to the current cart. They should also insert a row in the cart table for each inserted game.
2. Add the functionality to create the cart's content in **HomepageScripts.js**. You should create an empty cart function, and then create the cart table, which will be added to the `contentLoaded` function.

Results: The "Add to Cart" buttons on the Home, Game Details and Cart pages include the cart implementation.

Module Review and Takeaways

In the module, you learned about the client-side storage options that are available in HTML5. You also learned about web storage. The new client-side storage options open a lot of new possibilities to develop offline websites and applications, download resources to the client-side, and help to increase the performance of the site or application.

On the other hand, storing data on client-side can make websites and applications more vulnerable to hackers. Therefore, you should always think of data on client-side as public data. This should not discourage you from using client-side storage, but you should know this when you start using such storage.

Review Questions

Why is it so important to have client-side storage capabilities?

MCT USE ONLY. STUDENT USE PROHIBITED

Module 8

Using Advanced HTML5 JavaScript APIs

Contents:

Module Overview	8-2
Lesson 1: Using the Drag-and-Drop API	8-3
Lesson 2: File API	8-14
Lesson 3: Geolocation API	8-28
Lab: Using Advanced HTML5 JavaScript APIs	8-38
Module Review and Takeaways	8-42

Module Overview

- Using the Drag-and-Drop API
- File API
- Geolocation API

Along with new elements, attributes, and CSS3, HTML5 includes new and advanced JavaScript APIs that enable functionality that could not be done with current JavaScript. There are dozens of W3C Working Groups working on APIs such as Drag-and-Drop (DnD), working with files through the browser, geolocation to find users, and many other APIs. These APIs make many plug-ins such as Flash unnecessary because of the availability of relevant functionality already built inside the browsers. In the previous module, "Using Client-Side Storage", you were introduced to one such API and used it during the lab.

In this module, you will learn how to use other HTML5 JavaScript APIs, such as the drag-and-drop API, the file API to work with files, and the geolocation API to find the user location. There are many other APIs that the course does not cover, but you are encouraged to go to the W3C site and learn about other APIs. In addition, new APIs are being added from time to time, such as the Vibration API that was added while this course was being created.



Additional Reading: The HTML5 specifications and all the new JavaScript APIs can be found in the W3C site: [W3C Website](http://www.w3.org).

Objectives

After completing this module, you will be able to:

- Integrate drag-and-drop features into webpages.
- Work with the files by using the File API.
- Integrate the geolocation API into webpages.

Lesson 1

Using the Drag-and-Drop API

- Drag-and-Drop
- How to Enable Draggable Content
- Dragging Events
- dataTransfer Object
- Security Considerations

Drag-and-Drop (DnD) is becoming increasingly common behavior in websites and applications. It provides a richer experience in the site or application. Users increasingly expect sites and applications to provide such functionality for many components, for example, checkout carts in online purchasing. For years, developers who required this functionality in their site or application used external JavaScript libraries such as jQuery and jQuery UI plugin.

HTML5 introduces a new JavaScript API that enables drag-and-drop native browser capabilities. In this lesson, you will explore the new API and the additional markup attributes that enable element identification while performing drag and drop operations.

Lesson Objectives

After completing this lesson, you will be able to:

- Implement the new HTML5 Drag-and-Drop API.
- Integrate drag-and-drop functionality in a webpage.
- Implement dragging events.
- Describe the DataTransfer object.
- Describe the security considerations involved in using this API.

Drag-and-Drop

- For many years developers used external libraries to achieve drag-and-drop capabilities
- Drag-and-Drop API makes Drag-and-Drop functionality first class citizen in HTML5
- Using a set of events and objects enables every element in the DOM to be dragged and dropped



Drag-and-Drop is the ability to select an element by pressing and holding the primary mouse button on the element located on a webpage, dragging it to another element, and then dropping it by releasing the mouse button. This behavior is commonly used in many websites and applications because it helps users understand what they are doing, and it adds value to their overall experience. Examples for drag-and-drop can be dragging files to an upload surface on a webpage, such as dragging items into a checkout cart.

For years, developers were forced to use external JavaScript libraries to implement drag-and-drop capabilities. Those libraries used the built-in JavaScript events such as **mouseup** and **mousedown** to implement the behavior, requiring a lot of work behind the scenes to make it possible to drag-and-drop elements. If the developer wanted to use a proprietary behavior, it was very cumbersome.

HTML5 introduces a new Drag-and-Drop API. The API includes a set of events that can be wired-up, new markup attributes to indicate that an element can be dragged or dropped, and a new **DataTransfer** object to transfer data with the dragged element. All these enable the Drag-and-Drop functionality to be used to drag or drop any DOM element.

Having drag-and-drop functionality natively implemented in the browser improves the webpage performance because every browser vendor wants to optimize the performance. That means that using the API might help to create faster and more responsive websites and applications.

How to Enable Draggable Content

- Drag-and-drop API include two new HTML attributes:
 - draggable
 - dropzone
- Without these attributes drag-and-drop functionality does not work

```
<div draggable="true">The div can be dragged</div>
<div dropzone="copy">a dropzone</div>
```

The DnD API includes new markup attributes that enable the developer to express that an element can be dragged or dropped at a specific location. The attributes are **draggable** and **dropzone**. These attributes can be set on all DOM elements to enable every element on the webpage to be dragged or dropped at a specific location.

draggable: This attribute can be set on all DOM elements. The attribute has three optional values: *true*, *false*, and *auto*.

- The true value indicates that the element can be dragged.
- The false value indicates that the element cannot be dragged and is used mainly when an element is nested inside a draggable element.
- The auto value indicates that the browser default value (which might be true or false) is used as the attribute value.

The following example shows a div element with the draggable attribute.

draggable Attribute

```
<div draggable="true">The div can be dragged</div>
```

dropzone: This attribute is more complicated than the draggable attribute. This attribute has three operational values: copy, move, and link, and a set of unique tokens separated by spaces, which are used to filter the droppable data.

Operation values:

- **copy**—dragged data is copied.
- **move**—dragged data is moved to the new location, which is the dropzone.
- **link**—a link to the dragged data is created.

The following example shows a div element with the dropzone attribute.

dropzone Attribute

```
<div dropzone="copy">a dropzone</div>
```

Another option to set the dropzone is the set of unique tokens separated by spaces. The tokens can have two formats:

- Eight characters or more beginning with the "string:" string—indicates that the dragged data for the dropzone must be a string, where its type must match the string after the colon.
- Eight characters or more that begin with the "file:" string—indicates that the dragged data must originate from a file matching the type after the colon.

The following example shows a filter for file images of types png and jpg, which are set on a div elements dropzone:

dropzone Attribute with File Filter

```
<div dropzone="copy file:image/png file:image/jpeg"></div>
```

As shown in the preceding example, operations can be combined with filters. One restriction is that there can be only one operation (copy or move or link) in the **dropzone**. If no value is set to the **dropzone** attribute, the default value will be the copy value.

To enable drag-and-drop functionality, both attributes must be used. Omitting one attribute results in incomplete functionality. The only exception is when a drop target handles the **dragenter** event. In this case, you can omit the **dropzone** attribute.

Dragging Events

Drag-and-drop API includes a number of events to handle drag-and-drop:

- dragstart
- drag
- dragenter
- dragleave
- dragover
- drop
- dragend

```
var div = document.getElementById('draggableDiv');
div.addEventListener('dragstart', handleDragStart, false);
function handleDragStart(e) {
    this.style.opacity = '0.8';
}
```

The DnD API includes a set of events that can be wired-up with event handlers to perform drag-and-drop functionality. The DnD process starts with an origin element, the data that is going to be dropped, and a drop zone. The origin element can be any DOM element on the webpage, such as an image, link, block of HTML code, or even files. The data is the original element or any data payload to be dragged. The drop zone is the location where the data being dragged is dropped.

The following list describes the events that can be wired, as they are ordered in the DnD process:

1. **dragstart**—initializes the DnD process.
2. **drag**—called as long as the DnD process is running.
3. **dragenter**—called when dragged data is entering a drop zone.
4. **dragleave**—called when dragged data is leaving a drop zone.
5. **dragover**—called while dragged data is paused on a drop zone.
6. **drop**—called when the user releases the mouse button. It does not have to be inside a target drop zone.
7. **dragend**—called when the drop event is over. This is the last event in the DnD process.

To wire an event, use the **addEventListener** function on the element marked as **draggable**.

The following example shows how to wire the dragstart event to a div element.

Wiring DnD Events

```
var div = document.getElementById('draggableDiv');
div.addEventListener('dragstart', handleDragStart, false);
function handleDragStart(e) {
    this.style.opacity = '0.8';
}
```

In the code example, a **div** with an **id** of **draggableDiv** is retrieved. Then, it is wired to the **handleDragStart** event handler by using the **addEventListener** function. The **handleDragStart** event

handler changes the div opacity to 0.8. You can add other functionality when the drag is starting and the **dragstart** event is the place to add functionality. To change the opacity of the source element's back to its original state, use the **dragend** event. The same event wire-up process can be used with the other **DnD** events, enabling the developer to react to all the **DnD** process events.



Best Practice: In the drop event handler, use the `e.stopPropagation()` to prevent the default drop behavior of the browser. In the dragover event handler, use the `e.preventDefault()` to disable the browser behavior with links to navigate to a dragged link.

Some events can be wired on the drop zone. For example, the **dragover** and **drop** can be wired to the drop zone element.

dataTransfer Object

- The store to save the data which is transferred during the drag-and-drop process
- Part of the event exposed in the event handlers of the dragging events
- Include functions such as `getData` and `setData` to manipulate its content

```
var div = document.getElementById('draggableDiv');
div.addEventListener('dragstart', handleDragStart, false);
function handleDragStart(ev) {
    ev.dataTransfer.setData("text/plain", "stored text");
}
```

All the work being done in the DnD process is established in the **dataTransfer** object. Inside the event that is used in every **DnD** event handler, there is a **dataTransfer** object, which performs most of the DnD functionality. The **dataTransfer** object includes the data that is being sent during the drag event. The **dataTransfer** object can be set in the **dragstart** event handler and can be used in the **drop** event. The **dataTransfer** includes two main functions: **setData** and **getData**.

The **setData** function is used in the **dragstart** event handler to set the data to be dragged. The **setData** function has two parameters: *format* and *data*. The format is the *mimetype* used for the data and can include many options, such as **text/plain** to indicate a text format, or **text/html** to indicate HTML format. The data is used and stored in the **dataTransfer** property.

The following example shows a dragstart event handler that sets the data of the dataTransfer object.

Setting the Data of the dataTransfer Object

```
var div = document.getElementById('draggableDiv');
div.addEventListener('dragstart', handleDragStart, false);
function handleDragStart(ev) {
    ev.dataTransfer.setData("text/plain", "stored text");
}
```

In the preceding example, a draggable div is retrieved and then its **dragstart** event is wired to the **handleDragStart** function. The **handleDragStart** function sets the **dataTransfer** object data to the "stored text" string.

The **getData** function is in the **drop** event handler to retrieve data stored in the **dataTransfer** object. The **getData** function gets a format parameter to fetch the data by its *mimetype*.

The following example shows a drop event handler that gets the data of the dataTransfer object.

Getting the Data of the dataTransfer Object

```
var div = document.getElementById('dropDiv');
div.addEventListener('drop', handleDrop, false);
```



```
function handleDrop(ev) {
    var data = ev.dataTransfer.getData("text/plain");
    //add functionality to handle the data
}
```

In the preceding example, a div with the **dropzone** attribute is retrieved and then its drop event is wired to the **handleDrop** function. The **handleDrop** function gets the data out of the **dataTransfer** object. That data is retrieved as plain text.



Additional Reading: There are other functions that the course does not describe like **setDragImage**, **addElement**, and **clearData**. You can read about them in the following link: [Data Transfer](#)

The **DataTransfer** object also contains a set of properties that the developer can use. The most important properties are the **dropEffect** and **effectAllowed**. When the **dragenter** and **dragover** events are raised, the **dropEffect** holds the operational value that the DnD is going to perform (copy, link, move, or none). The developer can use the **dropEffect** to change the operation performed in these event handlers.

The following example sets the dropEffect during the dragover event handler.

dropEffect Property

```
var div = document.getElementById('dropDiv');
div.addEventListener('dragover', handleDragOver, false);
function handleDragOver(ev) {
    ev.dataTransfer.dropEffect = "copy";
    return false;
}
```

The **effectAllowed** property holds the permitted operations during the DnD process. These operations can be *none*, *copy*, *link*, *linkMove*, *move*, *all*, or *uninitialized*. The effect in this property is used to automatically set the **dropEffect** in the **dragover** and **dragenter** event handlers.

The following example shows how to set the effectAllowed property.

effectAllowed Property

```
var div = document.getElementById('draggableDiv');
div.addEventListener('dragstart', handleDragStart, false);
function handleDragStart(ev) {
    ev.dataTransfer.effectAllowed = 'copy';
    ev.dataTransfer.setData("text/plain", "stored text");
}
```

The preceding example shows how to enable only the copy effect when the **dragover** and **dragenter** event handlers are fired.



Additional Reading: For more information about the DnD specifications, you can go to the following link: [Drag and Drop Specifications](#)

Demonstration: Using the Drag-and-Drop API



This demonstration shows how to:

- Create a new Empty Web Application in VS2010
- Add a new HTML file.
- Add markup to the web page using the new HTML5 Drag-and-Drop API.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file's body element add the following elements:

```
<body>
  <div id="draggableDiv" draggable="true" style="width:150px; height:150px;border:
solid 1px black;">The div can be dragged</div>
  <div id="dropDiv" dropzone="copy" style="width:300px; height:300px; padding-
top:50px; border: solid 2px black;">a dropzone</div>
</body>
```

4. In the HTML file's head element add the following script tag:

```
<script type="text/javascript">
function wireDragEvents() {
  var div = document.getElementById('draggableDiv');
  div.addEventListener('dragstart', handleDragStart, false);
  var div = document.getElementById('dropDiv');
  div.addEventListener('drop', handleDrop, false);
  div.addEventListener('dragover', handleDragOver, false);
}
function handleDragStart(ev) {
  console.log('handling drag start event');
  ev.dataTransfer.effectAllowed = 'copy';
  ev.dataTransfer.setData("text", "stored text");
}
```

```
function handleDrop(ev) {
    if (ev.stopPropagation) ev.stopPropagation();
    console.log('handling drop event');
    var data = ev.dataTransfer.getData("text");
    console.log(data);
    return false;
}
function handleDragOver(ev) {
    if (ev.preventDefault) ev.preventDefault();
    console.log('handling drag over event');
    ev.dataTransfer.dropEffect = "copy";
    return false;
}
window.addEventListener("DOMContentLoaded", wireDragEvents, false);
</script>
```

5. Save the file as an HTML file.
6. In **Solution Explorer**, click on file with the right mouse button.
7. Click on the Set As Startup Page menu item.
8. Run the project by pressing CTRL+F5.
9. Drag the first div and drop it inside the second div.

Security Considerations

- Data in the **DataTransfer** object is available to scripts only after the drop event
- Drag-and-drop action must not start in response to script actions

Dragging and dropping elements and files might raise some security issues. The drag-and-drop specifications mostly deal with functionalities that should be implemented by the browser vendors. Such functionalities include:

- The **DataTransfer** object should not be available for use until the drop event occurs. Otherwise, third-parties that might intercept the object during the drag process could retrieve content that might be sensitive.
- If a script, instead of the user, ends the drag operation, browsers must consider the drag event unsuccessful and must not raise the drop event.
- Drag-and-drop functionality must not be started in response to a script action. It must always be started by a user.

Lesson 2

File API

- Working with Files
- File Objects
- Selecting Files
- Reading Files
- Security Considerations

One of the most common elements that developers deal with are files. Files are used for many tasks such as enabling the reading of saved data, saving data to a file, and manipulating files. Most of the development languages include APIs that help to perform these tasks.

JavaScript, on the other hand, did not include APIs for handling files, because it was considered very unsecure to enable websites and applications to read content on the client-side. This situation was changed when HTML5 introduced the **File API**, which can be used to select, read, and write to files. The API includes many new JavaScript objects and a set of functions that accompany the JavaScript objects. The API can be combined with other APIs such as **Drag-and-Drop** to achieve very complicated functionality.

In the lesson, you will learn what the **File API** is and how to use it to read files. The lesson does not discuss how to write to files or how to use file systems, because these HTML5 specifications are not yet finished.



Additional Reading: For more information about the File System API and the File Writer API, go to: [File System API](#) and [File Writer API](#).

Lesson Objectives

After completing this module, you will be able to:

- Describe how to work with files by using the **File API**.
- Describe File objects.
- Implement file selection.
- Implement file reading.

Working with Files

- File manipulation is a very common developer task
- Files are used for persisting and data reading
- Before HTML5 there was no way to read, manipulate and write files

Files are a basic element in operating systems like Microsoft® Windows®. Files can have numerous types and representations, such as text, binary, XML, and so on. Files help save data in formats for further use (this is sometimes called data persistence).

One of the basic tasks that junior developers learn is how to perform input/output (I/O) operations by using files. Such operations can be to read file content, append some content to an existing file, or create a new file. In most languages, there are APIs that make file reading and manipulation an easy task. JavaScript lacked this functionality.

HTML5 introduces the new **File API** that enables web developers to make I/O operations on files as it is done in other languages. The API is based on the asynchronous JavaScript behavior, and is, therefore, considered more complicated than other API implementations.

File API includes new file objects such as **File** and **Blob** to help the developer use the API. It also includes a set of functions to help read, write, and manipulate files. A new HTML functionality is added to the file input type to help in using multiple files at the same time. These changes in HTML5 makes functionalities such as dragging files into an HTML webpage or reading output from devices such as a webcam easier to achieve.

File Objects

The following objects are part of the File API specifications:

- Blob
- File
- FileList
- FileReader

```
var file = document.getElementById('file').files[0];

if (file) {
  var fileClone = file.slice();
  console.log(fileClone.name);
  console.log(fileClone.lastModifiedDate);
}
```

File API includes the following new JavaScript objects that help to achieve I/O functionality:

- **Blob**—represents immutable raw data stored in a file.
- **File**—holds typical file data such as name and size.
- **FileList**—is a list of one or more File objects.
- **FileReader**—reads the file content into memory by using a set of built-in functions.

Blob. The Binary Large Object (Blob) represents raw file content. The content is immutable and only for reading purposes. Blob includes two properties, *size* and *type* and one function, *slice*.

The *size* property is the Blob size in bytes. The size is used by other objects, such as **FileReader**, for reading purposes. If the Blob has no bytes to read, the size has a 0 value.

The *type* property is a lower case string that represents the Blob *media/MIME* type. Media/MIME types are the standard that identify the content type, for example, text, image, video, and other types. If the browser cannot identify the media type, it returns an empty string.

The *slice* function has three optional parameters—*start*, *end*, and *content* type. The *slice* function is used to slice Blob content into small Blob object pieces, to make it easier to read the Blob content. If the parameters are not used, the *slice* function returns an identical Blob object. The *start* and *end* parameters determine the returned, sliced Blob start point and end point. The content type parameter determines the returned Blob content type. The slice function occurs synchronously.



Note: The Blob slice function has the same name as the slice function of the **Array** and **String** JavaScript types. However, the Blob slice function is very different from the implementation of the JavaScript slice function.

The following example shows the slice function on a Blob object.

slice Function

```
// both of the calls retrieve an identical Blob object of the original blob
var blobClone = blob.slice();
var blobClone2 = blob.slice(0, blob.size);
// slice file into 1/2 chunk starting at beginning of file
var blobChunk = blob.slice(0, Math.round(blob.size/2));
```

In the preceding example, you can also see the `size` property, which is used for the slice calculation.

File. The **File** object represents a single file. It inherits the properties and the *slice* function from the **Blob** object and adds two new properties—*name* and *last modified date*. The **File** object is immutable. **File** objects are available only on objects that expose a *FileList* property type, such as a file input type.

The **name** property holds the file name without its path. If the browser cannot get the file name, the *name* property holds an empty string value.

The **lastModifiedDateproperty** holds the last modified date of the file. If the browser cannot determine the last modified date of the file, the property holds the null value.

The following example shows how to retrieve a File object and use the slice function it exposes.

File Object

```
// get the first file that is exposed by a file input type with the id file
var file = document.getElementById('file').files[0];
if(file)
{
    var fileClone = file.slice();
    console.log(fileClone.name);
    console.log(fileClone.lastModifiedDate);
}
```

The preceding example illustrates how to retrieve a file from a file input type, and then use the slice function to clone the file content. After the file is cloned, the file name and **lastModifiedDate** properties are logged to the console.

FileList. The **FileList** is a list of **File** objects. The **FileList** exposes the *length* property and the *item* function. The *length* property holds the number of files that exist inside the **FileList**, and if there are no files stored in the list, the 0 value. The *item* function gets an index as a parameter in the **FileList**, and returns the **File** object located in that index. If a file does not exist in the provided index, the function returns a null value.

There are currently two methods to get a **FileList**: the file input type, which has a *files* property, and the Drag-and-Drop **DataTransfer** object that includes the file property.

The following example retrieves a FileList object from a file input type with file id.

FileList Object

```
var fileList = document.getElementById('file').files;
```



Note: The **FileReader** object is discussed in the topic, "Reading Files"..

Selecting Files

- Files can be retrieved using the file input type
- HTML5 includes a new **multiple** attribute to indicate that several files can be selected
- Using Drag-and-Drop can also be used to select files

```
<input type="file" id="file" name="file" multiple />

function handleSelectedFiles(e) {
    var files = e.target.files;

    for (var i = 0; i < files.length; i++) {
        console.log(files.item(i).name);
        console.log(files.item(i).size);
        console.log(files.item(i).lastModifiedDate.toLocaleDateString());
    }
}

document.getElementById('file').addEventListener('change',
    handleSelectedFiles, false);
```

There are two methods to select files to work on: the file input type and the Drag-and-Drop **DataTransfer** object.



Note: Before selecting files, check whether the File API is supported in the browser. Unfortunately, the Modernizr library does not support such a check, so you will have to use your own check. Such a check can be written as: if (window.File && window.FileReader && window.FileList && window.Blob) {
}

The simplest way to load a file is by using the file input type. The file input type includes a **files** property that can be used to retrieve a **FileList** object. To enable the file input type to select more than a single file, HTML5 introduces the **multiple** attribute.

The following example shows a file input type with the multiple attribute.

Multiple Attribute

```
<input type="file" id="file" name="file" multiple />
```

Using the API objects and the file input type, developers can get files from the user and then manipulate the files.

The following example shows how to write to the console.log file properties, which are retrieved from the file input type, from the preceding code example.

Selecting Files with File Input Type

```
function handleSelectedFiles(e) {
    var files = e.target.files;
    for (var i = 0; i < files.length; i++) {
        console.log(files.item(i).name);
        console.log(files.item(i).size);
    }
}
```

```

        console.log(files.item(i).lastModifiedDate.toLocaleDateString());
    }
}
document.getElementById('file').addEventListener('change', handleSelectedFiles, false);

```

The example defines the **handleSelectedFiles** function that is wired to the file input type change event. The function gets the **FileList** from the raised event and outputs the file's information to the console.

Another option to retrieve files is by using the Drag-and-Drop **DataTransfer** object, which was presented in the first lesson of this module.

The following example shows how to use the DataTransfer object to achieve the same results as in the preceding code example.

Selecting Files by Using the Drag-and-Drop API

```

<!-- This code belongs in the HTML file -->
<div id="dropzone" dropzone="dropzone">Drop files into this location</div>
//This code belongs in the JavaScript file
// the callback function that will be called in the drop event
function handleSelectedFiles(e) {
    e.stopPropagation();
    e.preventDefault();
    var files = e.dataTransfer.files;
    for (var i = 0; i < files.length; i++) {
        console.log(files.item(i).name);
        console.log(files.item(i).size);
        console.log(files.item(i).lastModifiedDate.toLocaleDateString());
    }
}
// the callback function that will be called in the dragover event
function handleDragOver(e) {
    e.stopPropagation();
    e.preventDefault();
    e.dataTransfer.dropEffect = 'copy';
}
// Setup the drag-and-drop event listeners.
var dropZone = document.getElementById('dropzone');
dropZone.addEventListener('dragover', handleDragOver, false);
dropZone.addEventListener('drop', handleSelectedFiles, false);

```

The preceding code is divided into two sections. The first section is a **div** definition that is used in HTML to define a drop zone for the Drag-and-Drop behavior. The second section is the JavaScript code that defines two event handlers: **handleDragOver** and **handleSelectedFiles** wired to the **dragover** and **drop** events. This code resembles the previous code example, but instead of using the file input type, it uses the **DataTransfer** object.

Reading Files

- Reading files is achieved by the `FileReader` object
- The reading operation is asynchronous
- Use several API functions to read:
 - `readAsText`
 - `readAsDataURL`
 - `readAsArrayBuffer`

```
function handleSelectedFiles(e) {
  var files = e.target.files;
  for (var i = 0; i < files.length; i++) {
    var reader = new FileReader();
    // Closure to capture the file information
    reader.onload = (function (theFile) {
      return function (e) {
        console.log(e.target.result);
      };
    })(files[i]);
    reader.readAsText(files[i]);
  }
}
```

After you select a file and have it represented by the `File` or `Blob` objects, the file content can be read. To read the file, use the **FileReader** object to read the objects into memory, and access their content. The reading process is asynchronous, enabling wiring event handlers to monitor the reading progress.

The **FileReader** object exposes a lot of functionality, including three properties: *readyState*, *result*, and *error*.

The **FileReader** object also includes the following reading functions to read the file content:

- `readAsText`
- `readAsDataURL`
- `readAsArrayBuffer`

The **FileReader** object includes another function, called *abort*, to abort the reading process.

The **FileReader** object also includes the following events that can be wired with handlers to monitor the reading progress:

- **onloadstart**
- **onprogress**
- **onload**
- **onabort**
- **onerror**
- **onloadend**

The **FileReader** Properties

readyState. This property holds the following three different values:

- **EMPTY**—represented by the 0 value, which means that none of the reading functions were called on the **File** or **Blob** object.

- **LOADING**—represented by the 1 value, which means that the **File** or **Blob** is being read.
- **DONE**—represented by the 2 value, which means that the **File** or **Blob** reading is ended.

result. This property holds the reading result of the **File** or **Blob** object. The property can return partial Blob data if it is being called before the file reading ends. When the *readystate* is **EMPTY** or an error occurs in the reading process, the **result** holds the null value. Otherwise, the **result** returns the file content in the same format that the reading function returned it.



Note: If the content reading process is successful, the **result** property returns a non-null value.

error. This property hold a value only if an error occurs during the reading process. The error can be one of the following types:

- **NotFoundError**—the File or Blob is not found during the reading process. That might happen if the user erased the file or moved the file to another location.
- **SecurityError**—the File or Blob is unsafe to read. The user started too many read calls, or the file was changed during the reading process.
- **NotReadableError**—the File or Blob cannot be read by the FileReader.
- **EncodingError**—the **readAsDataURL** call encountered an error. Data URL has a length limitation, and if exceeded, this error type is thrown in the reading process.

The FileReader Reading Functions

readAsText. The function reads the File or Blob content as text. The function gets two parameters, a *Blob* and an *optional encoding*. When the function is called, the following outcomes are possible:

- If the *readystate* is **LOADING**, an exception is thrown.
- If an error occurs, the *readystate* becomes **DONE**, the result property is set to null, and the error property holds the error.
- No error occurred; the file is being read as text asynchronous. At the end of the process, the result property holds a text representation of the file content and the *readystate* is **DONE**.

The following example shows how to read a file obtained from a file input type by using the **readAsText**.

Reading a File by Using readAsText

```
function handleSelectedFiles(e) {
    var files = e.target.files;
    for (var i = 0; i < files.length; i++) {
        var reader = new FileReader();
        // Closure to capture the file information
        reader.onload = (function(theFile) {
            return function(e) {
                console.log(e.target.result);
            };
        })(files[i]);
        // Read the file as a text.
        reader.readAsText(files[i]);
    }
}
document.getElementById('file').addEventListener('change', handleSelectedFiles, false);
```

readAsDataURL. The function reads the File or Blob content encoded as a **Data URL**. Data URL is a scheme that provides a way to include data inline in a webpage, as if it is an external resource. Data URL can help reduce the number of HTTP requests for resources in the page, but it increases the page size. Its main use is to embed small images encoded as a Data URL, which are put as a source to the **** tag. Data URL are encoded in base-64 and are written in a specific format. The format starts with the data string, followed by the mime type of the data. After the mime type, a base64 string is written and then all the encoded data is written.

The following example shows a Data URL as a source of an image tag.

Data URL Example

```

```

In the preceding example, an image element gets a data URL representing a red dot, as a source.

Data URLs have a size limitation, which means that trying to read a big file as a Data URL might raise an **EncodingError** exception.

When the function is called, the possible outcomes are as follows:

- If the *readystate* is **LOADING**, an exception is thrown.
- If an error occurs, the *readystate* becomes **DONE**, the result property is set to null, and the error property holds the error.
- No error occurs and the file is being read as Data URL asynchronously. At the end of the process, the result property holds a text representation of the file content, and the *readystate* is **DONE**.



Note: Data URL without media types are considered as plain text by the browser.

The following example shows how to read a file obtained from a file input type by using the `readAsDataURL`.

Reading a File by Using `readAsDataURL`

```
function handleSelectedFiles(e) {
    var files = e.target.files;
    for (var i = 0; i < files.length; i++) {
        var reader = new FileReader();
        // Closure to capture the file information
        reader.onload = (function(theFile) {
            return function(e) {
                console.log(e.target.result);
            };
        })(files[i]);
        // Read the file as a text.
        reader.readAsDataURL(files[i]);
    }
}
document.getElementById('file').addEventListener('change', handleSelectedFiles, false);
```

readAsArrayBuffer. The function reads the File or Blob content as an **Array Buffer** object. The array holds the file content, which can then be manipulated by setting the array.

When the function is called, the following outcomes are possible:

- If the *readystate* is **LOADING**, an exception is thrown.
- If an error occurs, the *readystate* will be **DONE**, the result property is set to null, and the error property holds the error.
- No error occurs and the file is being read as Data URL asynchronous. At the end of the process, the result property holds a text representation of the file content, and the *readystate* is **DONE**.

The following example shows how to read a file obtained from a file input type by using the `readAsArrayBuffer`.

Reading a File by Using `readAsArrayBuffer`

```
function handleSelectedFiles(e) {
    var files = e.target.files;
    for (var i = 0; i < files.length; i++) {
        var reader = new FileReader();
        // Closure to capture the file information
        reader.onload = (function(theFile) {
            return function(e) {
                console.log(e.target.result);
            };
        })(files[i]);
        // Read the file as a text.
        reader.readAsArrayBuffer(files[i]);
    }
}
document.getElementById('file').addEventListener('change', handleSelectedFiles, false);
```

abort. The function terminates the file reading execution. The function is used to perform the following actions:

- If *readystate* is **EMPTY** or **DONE**, the result property is set to null.
- If *readystate* is **LOADING**, it is set to **DONE**, and the result property is set to null.
- It then fires the **abort** and **loaded** event one after the other.

The `FileReader` Events

As previously described, `FileReader` exposes numerous events such as **onloadstart**, **onprogress**, **onload**, **onabort**, **onerror**, and **onloadend**. These events can be wired-up to event handlers and help monitor the reading progress. Because the reading operation is asynchronous, using the handlers can help the developer to monitor for large files reading, catch any errors during the reading process, and determine when a read is complete.

The following example shows how to wire the events to write some information to `console.log`.

Using `FileReader` Events

```
reader.onabort = function(e) {
    Console.log('File read cancelled');
};
reader.onloadstart = function(e) {
    console.log('loading the file');
};
```

In the preceding example, the reader is a `FileReader` object.



Note: For more information about File API events, you can go to the following link:
<http://go.microsoft.com/fwlink/?LinkID=241716>

Demonstration: File API



This demonstration shows how to:

- Create a new Empty Web Application in VS2010.
- Add a new HTML file.
- Add markup to the web page using the new HTML5 File API.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application.
2. Create an HTML file.
3. In the HTML file's body element add the following elements:

```
<body>
  <input type="file" id="file" name="file" multiple />
</body>
```

4. In the HTML file's head element add the following script tag:

```
<script type="text/javascript">
  function handleSelectedFiles(e) {
    var files = e.target.files;
    for (var i = 0; i < files.length; i++) {
      var reader = new FileReader();
      reader.onload = (function (theFile) {
        return function (e) {
          console.log(e.target.result);
        };
      })(files[i]);
      // Read the file as a text.
      reader.readAsText(files[i]);
    }
  }
  function contentLoaded() {
```



```
        document.getElementById('file').addEventListener('change',  
        handleSelectedFiles, false);  
    }  
    window.addEventListener("DOMContentLoaded", contentLoaded, false);  
</script>
```

5. Save the file as an HTML file.
6. In Solution Explorer, click on file with the right mouse button.
7. Click on the Set As Startup Page menu item.
8. Run the project by pressing CTRL+F5.
9. Press the Browse button and pick any text file to see the result of the reading in the Visual Studio's console view.

Security Considerations

- Reading files from Internet applications might be considered a security breach
- File API specifications demand a lot of restrictions that the browsers must supply
- The File API is considered secured



For years, browsers did not enable reading or writing files to avoid creating security vulnerability. The idea that an Internet site could read files in the user's file system was frightening. This is why the File API creates a sandbox environment to read and write files. The File API cannot read files in the user's file system without the user's permission.

According to the HTML5 specifications, browsers are supposed to prevent attacks such as:

- **Preventing selection looping**—where the user is stuck within a loop that forces a file selection in a file input type.
- **System-sensitive files**—sensitive files are not exposed to websites and applications. Such files can be the operation system files or other sensitive file types. If secured files are being used with the File API, a **SecurityError** exception is thrown.
- **Cross-origin requests on Blob URIs**—Blobs cannot be accessed across origins.

There are other attacks written in the specifications and more will be added in the future to create a more secured interface.

 **Additional Reading:** For more information about File API security, you can go to the following link: [File API Security](#)

Lesson 3

Geolocation API

- What Is Geolocation?
- Geolocation Functions
- Geolocation Objects
- Security Considerations

The Geolocation API is probably one of the most interesting APIs in the HTML5 JavaScript APIs. *Geolocation* is an abbreviation for *geographic location*. This API enables the browser to retrieve the geographic location of the user, which can include not only his or her position, but also information such as heading and speed. The Geolocation API is very useful for websites and applications that are built for use by mobile phones or tablets, with content that depends on the location of the user.

The API includes three functions and four main objects. The API also includes strong restrictions regarding the privacy of the user. These restrictions must be built into the browser to deny any abuse of the API. In this lesson, you will learn about the Geolocation API and how to apply it in your webpages.

Lesson Objectives

After completing this module, you will be able to:

- Apply the Geolocation API in webpages.

What Is Geolocation?

- The ability to retrieve the geographic location of a user
- An API that retrieves location information associated with the user
- Based on the user's device



Geolocation is the ability to locate the geographic position of the user, based on a set of API functions. In the development world, there are many web-capable devices that incorporate a built-in Global Positioning System (GPS) that can be used to locate a user. There are other means of retrieving a user's location: through his or her IP address, for instance, or through WiFi and Bluetooth MAC addresses. GPS can be extremely accurate, but the other means are not necessarily so, and they usually given an approximation of the user's location.

The API can be used to obtain the location once or periodically. The user can deny permission to be located by the API—which browsers should use as the default behavior.



Note: The API does not guarantee that information on the location of the user will be accurate. The results of running the Geolocation API should therefore not be trusted.

Geolocation Functions

- `getCurrentPosition`
- `watchPosition`
- `clearWatch`

```
navigator.geolocation.getCurrentPosition(successCallback, errorCallback);

function successCallback(e) {
    // do something when the location retrieval succeeds
}

function errorCallback(e) {
    // do something when the location retrieval fails
}
```

The Geolocation API includes three functions that are exposed by the **window.navigator.geolocation** object. The functions are:

- **getCurrentPosition**
- **watchPosition**
- **clearWatch**

getCurrentPosition. Enables the retrieval of the user's location only once. It receives three parameters—success callback, error callback, and a **PositionOption** object. If the location retrieval succeeds, the success callback function will be called; if it fails, the error callback will be called.

The following example shows how to use `getCurrentPosition` with success and error callbacks.

The `getCurrentPosition` Function

```
navigator.geolocation.getCurrentPosition(successCallback, errorCallback);
function successCallback(e) {
    // do something when the location retrieval succeeds
}
function errorCallback(e) {
    // do something when the location retrieval fails
}
```

watchPosition. Listens continually for updates to the current geographical location of the user. Has the same signature as `getCurrentPosition`, but returns a watch id that will later be used to clear the watch. The watch position function raises its callback functions every time the position of the device changes, enabling the browser to track the heading and speed of the user.

The following example shows how to use `watchPosition` with success and error callbacks.

The `watchPosition` Function

```
var watchid = navigator.geolocation.watchPosition(successCallback, errorCallback);
function successCallback(e) {
```

```
// do something when the location retrieval succeeds
}  
function errorCallback(e) {  
    // do something when the location retrieval fails  
}
```

clearWatch. Stops listening for updates to the current geographical location that had been sent since the activation of the **watchPosition** function. It receives as parameter the watch id returned by the **watchPosition** function. If the supplied watch id is not associated with any current **watchPosition** process, nothing will happen.

The following example shows how to use the **clearWatch** function.

The **clearWatch** Function

```
navigator.geolocation.clearWatch(watchid);
```

Geolocation Objects

- **PositionOptions**; object to configure the request for the user's location
- **Position**; object, which is returned in every position success callback
- **Coordinates**; object, which is nested inside the Position object
- **PositionError**; object which contains error data when the Geolocation operation fails

```
function successCallback(position) {
    console.log(position.coords.latitude);
    console.log(position.coords.longitude);
}
```

The Geolocation API uses four different JavaScript objects during the process of retrieving positional data:

- **PositionOptions**
- **Position**
- **Coordinates**
- **PositionError**

PositionOptions. This object is used as the third parameter when calling the **getCurrentPosition** and **watchPosition** functions. The **PositionOptions** object helps the functions configure the request with the properties that it includes, which are:

- **enableHighAccuracy**
- **timeout**
- **maximumAge**

enableHighAccuracy is a Boolean value that indicates whether the application wants to receive the most accurate results. This, of course, does not guarantee that the result will be accurate. Setting this parameter to *true* can slow down the acquisition of the positional data. This Boolean is mainly used with a *false* value, which indicates to the Geolocation implementation that the speed of the response takes priority over the accuracy of the result. The default value for the property is *false*.

timeout is an integer that provides the maximum amount of time, in milliseconds, for the browser to obtain the location. If the timeout expires and the device is unable to successfully acquire the user's location, it will raise the error callback. If the **PositionOptions** object is not supplied when **getCurrentPosition** or **watchPosition** are called, the operation will not timeout. If a negative or zero value is supplied, the timeout value is considered 0.



Note: The period of time required to obtain the user's permission to be located is not included in the timeout.

maximumAge is an integer value in milliseconds that expresses the maximum age of any cached position information that the website or application will accept as a valid response. Devices may cache their positions to achieve better performance by having a ready response. If the value provided in **maximumAge** is more than the age in milliseconds of an available cached position, that cached position will be used; otherwise, the device will fetch a new position. If a value of 0 is provided, the device will be asked to return a new position object. If the value is set to Infinity, any cached value available will be used. The default value for the property is 0.

The following example shows how to use the **PositionOptions** object with the **getCurrentPosition** function.

The PositionOptions Object

```
navigator.geolocation.getCurrentPosition(successCallback, errorCallback,
{
  enableHighAccuracy: true,
  maximumAge: 1000,
  timeout: 500
});
function successCallback(e) {
  // do something when the location retrieval succeeds
}
function errorCallback(e) {
  // do something when the location retrieval fails
}
```

The preceding example uses the literal notation of the JavaScript object to create the **PositionOptions** object and set it inside the call for the function.

Position. Used as a container for the result of a successful retrieval of a new position, it includes two properties: a **Coordinates** object and a timestamp. The timestamp contains the time the **Position** object is acquired; the **Coordinates** object is represented by the **coords** property.

The following example shows a success callback that writes the timestamp of the position to **console.log**.

The Position Object

```
function successCallback(position) {
  console.log(position.timestamp);
}
```

Coordinates. This is a container for all the coordinate data that is retrieved. **Coordinates** contains the following properties:

- **latitude**—the latitude of the geographic coordinate, in decimal degrees.
- **longitude**—the longitude of the geographic coordinate, in decimal degrees.
- **altitude**—the height from sea level of the position, in meters.
- **accuracy**—the accuracy of the position, in meters.
- **altitudeAccuracy**—the accuracy of the altitude, in meters.
- **heading**—the direction of travel, specified in degrees from 0 to 360 degrees clockwise.
- **speed**—the velocity of the device, specified in meters per second.

Not all of these properties necessarily contain a value when the success callback is called; this depends on the device that is used.

The following example shows how to write some of the values of the **Coordinates** object to **console.log**.

The Coordinates Object

```
function successCallback(position) {  
    console.log(position.coords.latitude);  
    console.log(position.coords.longitude);  
}
```

PositionError. When an error occurs, the error callback is called with the **PositionError** object. This object includes two properties—**code** and **message**. The **code** includes the following error code options:

- **PERMISSION_DENIED**—indicates that the location process failed because the user denied permission. Its value is equal to 1.
- **POSITION_UNAVAILABLE**—indicates that the location could not be determined. Its value is equal to 2.
- **TIMEOUT**—indicates that the operation to retrieve the location timed out. Its value is equal to 3.

The **message** property describes the details of the error encountered.

The following example shows an error callback function that uses the **PositionError** object.

The PositionError Object

```
function errorCallback(error) {  
    var message = "";  
    switch (error.code) {  
        case error.PERMISSION_DENIED:  
            message = "Permissions denied by the user ";  
            break;  
        case error.POSITION_UNAVAILABLE:  
            message = "The current position could not be determined.";  
            break;  
        case error.PERMISSION_DENIED_TIMEOUT:  
            message = "timeout occurred ";  
            break;  
    }  
    console.log(message);  
}
```

Demonstration: Geolocation API



This demonstration shows how to:

- Create a new Empty Web Application in VS2010.
- Add a new HTML file.
- Add markup to the web page using the new HTML5 Geolocation API.

Demonstration Steps

1. Create a new C# ASP.NET Empty Web Application
2. Create an HTML file
3. In the HTML file's head element add the following script tag:

```
<script type="text/javascript">
navigator.geolocation.getCurrentPosition(successCallback, errorCallback,
{
    enableHighAccuracy: true,
    maximumAge: 1000,
    timeout: 500
});
function successCallback(position) {
    console.log(position.coords.latitude);
    console.log(position.coords.longitude);
}
function errorCallback(error) {
    var message = "";
    switch (error.code) {
        case error.PERMISSION_DENIED:
            message = "Permissions denied by the user ";
            break;
        case error.POSITION_UNAVAILABLE:
            message = "The current position could not be determined.";
            break;
        case error.PERMISSION_DENIED_TIMEOUT:
            message = "timeout occurred ";
            break;
    }
}
```

```
    }  
    console.log(message);  
  }  
</script>
```

4. Save the file as an HTML file.
5. In Solution Explorer, click on file with the right mouse button.
6. Click on the Set As Startup Page menu item.
7. Run the project by pressing CTRL+F5.
8. You should press the Allow Once button that is popped up by the browser.

Security Considerations

- Retrieving the location of a user potentially compromises the user's privacy
- The API requires the user's permission to locate his position
- Without the user's permission, every Geolocation request should result in an error



Retrieving the user's location can potentially compromise the user's privacy. It is the browser vendor's responsibility to request the user's permission to get his location. If the user refuses, the Geolocation retrieval will fail and call the error callback with the `PERMISSION_DENIED` error. The browser vendor is also responsible for implementing easy access to interfaces that enable the revocation of permissions, which allows users to deny permission after it has already been granted by mistake.

Lab: Using Advanced HTML5 JavaScript APIs

- Integrating Drag-and-Drop Behavior into the Home Page
- Add Geolocation Information to the Purchase Event
- Add Coupon-Reading Behavior

Logon Information

Virtual Machine	Virtual machine: SEA-DEV
User Name	User name: Admin
Password	Password: Pa\$\$w0rd

Estimated Time: 60 minutes

Scenario

The Contoso, Ltd. management requires enhanced shopping cart functionality in various ways. You have been asked to improve the Home Page experience by adding drag-and-drop capabilities to enable users to drag games from the game catalog into the shopping cart.

When customers buy games, they should be able to use coupons in the form of text files to get a discount. The management also wants the location of each user to be sent server-side during the purchasing process.

Objectives

In this lab you will:

- Modify the home page and integrate drag-and-drop behavior.
- Provide basic knowledge about how to use advanced JavaScript APIs such as the File API and the Geolocation API.
- Implement file-reading and location-detail behavior in the online store.

Exercise 1: Integrating Drag-and-Drop Behavior into the Home Page

Scenario

You are asked to enable users to drag games on the Home Page from the Games catalog to the shopping cart by using the new HTML5 Drag-and-Drop API.

► Task 1: Integrate Drag-and-Drop Behavior.

1. Open the starter solution, which is located in the **D:\Mod08\Labfiles\Starter** folder.
2. Add the **addDragAndDropEventHandlers** function, which handles the **DnD** events wire-up.
3. Add the **draggable** attribute to the game elements.
4. Add the **dropzone** attribute to the **cart** section.

Results: A Home Page with drag-and-drop behavior that enables the user to drag games from the Games catalog and drop them into the shopping cart.

Exercise 2: Add Geolocation Information to the Purchase Event

Scenario

You are asked to integrate a Geolocation call to get the latitude and longitude of the client, and send it to the server.

► Task 1: Create the Purchase Event.

1. Create the **addPurchaseEvent** function.
2. Add the **Location** object to **GlobalScripts.js**.
3. Check whether Geolocation is enabled. If it is enabled, call the **getCurrentPosition** function.
4. Add the **makePurchase** function.

Results: The purchase data that is sent to the server contains the location data.

Exercise 3: Add Coupon-Reading Behavior

Scenario

You are asked to enable the reading of a coupon on the checkout page. First, integrate a file input type and then read its content in the purchase event.

► Task 1: Add a file input type to the Checkout webpage.

1. Add a **file** input type to the Checkout webpage.

► Task 2: Add the Coupon-Reading Functionality.

1. Add a **readCoupon** function to handle the coupon reading.

Results: When the Purchase button is clicked, an alert with the text data in the coupon text file is shown on screen.

Module Review and Takeaways

This module introduced three new HTML5 JavaScript APIs that you can use to create richer and better websites and applications:

- The Drag-and-Drop API enables you to drag elements on a webpage and drop them into drop zones.
- The File API enables you to select and read files.
- The Geolocation API enables you to use the browser to locate and communicate the user's geographic location.

Many other HTML5 JavaScript APIs are available, and you should explore them on the W3C website.

Module 9

Using WebMatrix and Other Developer Tools

Contents:

Module Overview	9-2
Lesson 1: WebMatrix	9-4
Lesson 2: ASP.NET MVC 3 and Razor	9-9
Lab: Using WebMatrix and Other Development Tools	9-18
Module Review and Takeaways	9-21

Module Overview

- WebMatrix
- ASP.NET MVC 3 and Razor

The Microsoft® web platform provides a rich set of tools that help to develop modern websites and applications by using HTML5. The Microsoft web platform includes tools designed for different types of programmers with varying levels of experience. Two of the tools created and published by Microsoft are Microsoft WebMatrix® and Microsoft Visual Studio® 2010.

- WebMatrix is a free development tool that includes everything needed for website development. It can be installed in minutes and it brings together a web server, a database, and programming frameworks into a single, integrated experience.
- Visual Studio 2010 is an Integrated Development Environment (IDE) and home for many integrated tools that developers use in everyday tasks. One of the integrated technologies is ASP.NET MVC 3, which provides a framework for building scalable, standards-based web applications by using well-established design patterns and the power of ASP.NET and the Microsoft .NET Framework.

In this module, you will be introduced to the tools and will learn how to use them to create websites and applications. Since HTML5 is a client technology and can't stand alone in most of the websites and applications without a server-side implementation, it is important to learn about the Microsoft® web platform tools.



Note: In real world scenarios there are no such things as static websites and web applications. Most of the websites and web application use some sort of server technology in order to create their desirable functionality. The Microsoft® web platform provides WebMatrix and ASP.NET MVC as their main server technologies for the creation of websites and web applications. Both, WebMatrix and ASP.NET MVC, enable developers to use HTML5 as their client-side technology.

Objectives

After completing this module, you will be able to:

- Use WebMatrix.
- Use ASP.NET MVC 3 and the Razor view engine.

Lesson 1

WebMatrix

- Installing and Configuring WebMatrix by Using the Web Platform Installer
- Getting to Know WebMatrix

This lesson introduces WebMatrix, which helps to build fully customizable and powerful websites by using virtually any web technology. It supports pure HTML development along with technologies such as ASP.NET, PHP, and others. In this lesson, you will become familiar with the WebMatrix IDE and its project structure and product features. In addition, you will learn about WebMatrix features such as managing databases, using IIS Express to host your website locally and manage its virtual directories, and using the Web Platform Installer to install WebMatrix and supported products.

Lesson Objectives

After completing this lesson, you will be able to:

- Install and configure WebMatrix by using the Web Platform Installer.
- Use WebMatrix.

Installing and Configuring WebMatrix by Using the Web Platform Installer

- The Web Platform Installer installs the Microsoft Web Platform components
- Supports popular free 3rd party Web applications
 - Provided through the Web Application Gallery
 - Applications include free blog engines, content management, and more
- Tiny download
 - Web Platform Installer is around 2Mb of download size
- It is free



WebMatrix is a free tool that can be installed by using the Web Platform Installer. The Web Platform Installer (Web PI) is a free tool that enables easy and convenient installation of Microsoft Web Platform components and configuration requirements needed to successfully run new installations. Products supported by the Web Platform Installer include Internet Information Services (IIS) web server and IIS Express (light edition of IIS), Microsoft SQL Server Express, the .NET Framework, Visual Web Developer tools, and others. The Web Platform Installer also makes it easy to install and run the most popular free web applications for blogging, content management, and more, with the built-in Web Application Gallery.



Note: The Web Platform Installer can be downloaded from the following location:
<http://go.microsoft.com/fwlink/?LinkID=145505>

The Web Platform Installer is free to download and is very small, with only 2 megabytes (MB) download size. After installing and running the Web Platform Installer, you will be presented with a products screen that enables you to configure and install the supported products. In some cases, installing one product requires additional components to be installed. The Web Platform Installer checks the selected product's dependencies and installs the required components automatically.

Getting to Know WebMatrix



What is WebMatrix?

WebMatrix is a free and lightweight IDE for web development that helps to build websites by using virtually any web technology. WebMatrix provide database support, integrate with open source web applications such as WordPress and Drupal, and include easy deployment support. Using the WebMatrix helps to create and deploy websites and web applications much faster since it puts everything that a web developer need in one place.

Why and When to Use WebMatrix?

WebMatrix is a free tool that you can download using the Web Platform Installer or from the WebMatrix website. It helps to rapid develop websites and web applications and it integrates many open source web applications that can be customized using the tool. WebMatrix also include the ability to deploy the created website or application faster and easier.

Getting Started with WebMatrix

WebMatrix is designed for fast and easy website creation. Using WebMatrix, you can create a new site from one of the built-in templates, or just open an existing site and edit the source content. Whatever you do, WebMatrix provides you with everything your website needs, including the web server, database, and developer framework.

After you install and start WebMatrix, you are presented with a Welcome screen. This enables you to create a new WebMatrix project from the online website gallery, from one of the built-in templates, or from an existing folder. You can also open an existing WebMatrix project by using the **My Sites** icon.

The online website gallery and built-in templates provide a convenient place to start building a new website. Opening a site from an existing folder provides an easy way to upgrade an existing website with new functionality.

In all cases, after you select the most suitable starting point, WebMatrix sets up a new virtual directory in IIS Express (installed with WebMatrix) and provides an easy and intuitive interface to configure various settings for the new site.



Note: A vital part for any website is a web server, which hosts the files and enables access to them from the World Wide Web (WWW). WebMatrix is capable of working with Microsoft Internet Information Services (IIS) and IIS Express (light edition of IIS) to publish the site to virtually any remote web server by using FTP or Microsoft WebDeploy protocols.



Note: IIS Express is a light edition of IIS (Internet Information Services web server), which is used to mimic IIS web server for development purposes.



Note: File Transfer Protocol (FTP) is a standard network protocol used to transfer files from one host to another host over a TCP-based network, such as the Internet. Microsoft WebDeploy protocol is a free technology developed by Microsoft and used to improve the website deployment process by including built-in support for creating virtual directories and applications within IIS, copying files and folders, provisioning databases, registering components, and so on.

IIS and IIS Express both use a *virtual directory* to map between a local folder on the server machine and a web directory name (sometime referred to as *path*). The directory name then becomes part of the site URL, and users can request the URL from a browser to access content in the physical folder, such as a webpage or a list of additional folders and files.



Best Practice: It is a good practice to specify a different name for the virtual directory than the physical folder. Doing so makes it more difficult for users to discover the actual physical file structure on your server if the URL does not map directly to the root of the site.

The WebMatrix UI has several sections:

- The ribbon provides easy access to available commands and actions.
- In the Navigation pane on the left, you can move among the pages of your website.
- In the Workspace selector, you can switch among the different workspaces, such as Site, Files, Databases, and Reports.
- The Content pane changes are based on the workspace and page you are working on.

Switching to a different workspace displays different commands on the ribbon. For example, switching to the Files workspace enables you to add a new file or folder to the site.

The Files Workspace

WebMatrix supports many file formats that help to develop the website according to the selected technology. The supported file formats include HTML, CSS, CSHTML, and others.



Note: CSHTML is a file extension that is used to indicate a Razor file. You will learn about the Razor view engine in lesson 2.

After you select a new file and add it to the site, WebMatrix provides syntax highlighting and suggestions to help you quickly develop the site.

The Database Workspace

In the Database workspace, you can create a new database, define new tables and relations, and edit table data.

The database is an organized collection of data. The data is typically organized to model relevant aspects of reality (for example, game titles in a store), in a way that supports processes that require this information (for example, finding the title requested by the user). This definition is very general, and is independent of the technology used. In our case, the database is a computer database, which is a structured collection of records or data that is stored in a computer system.

 **Note:** WebMatrix supports using databases on various versions of Microsoft SQL Server, including Microsoft SQL Server Express Edition. SQL Server Express is free to install and use, and it can be installed as part of the installation process of the Web Platform Installer. By default, SQL Server Express is installed with WebMatrix.

In addition, the Database workspace also enables you to connect to existing databases on computers that run SQL Server.

The Reports Workspace

The Reports workspace lets you run site analysis reports to improve site performance and fix errors.

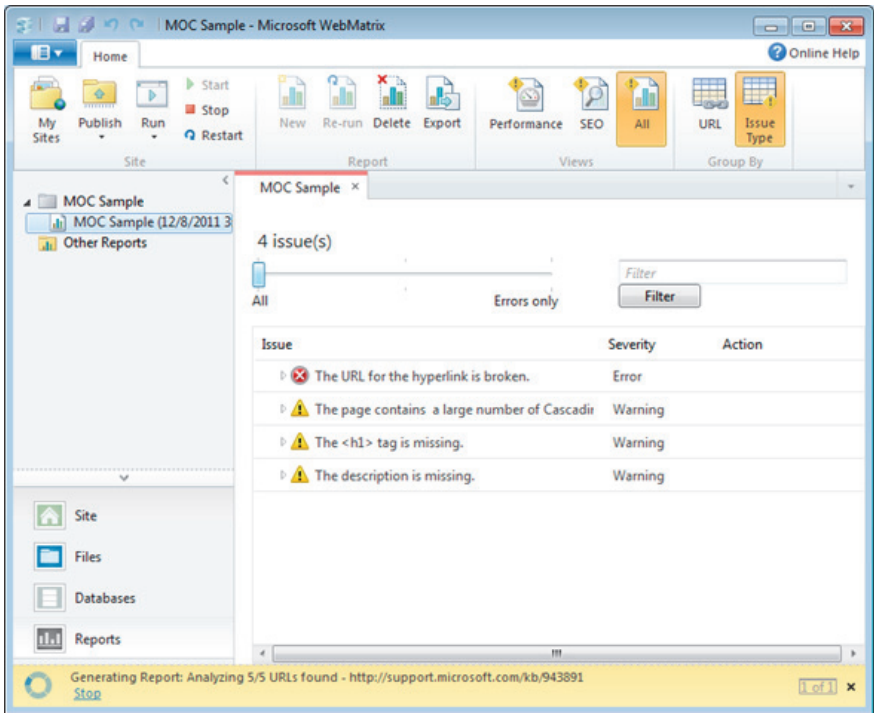
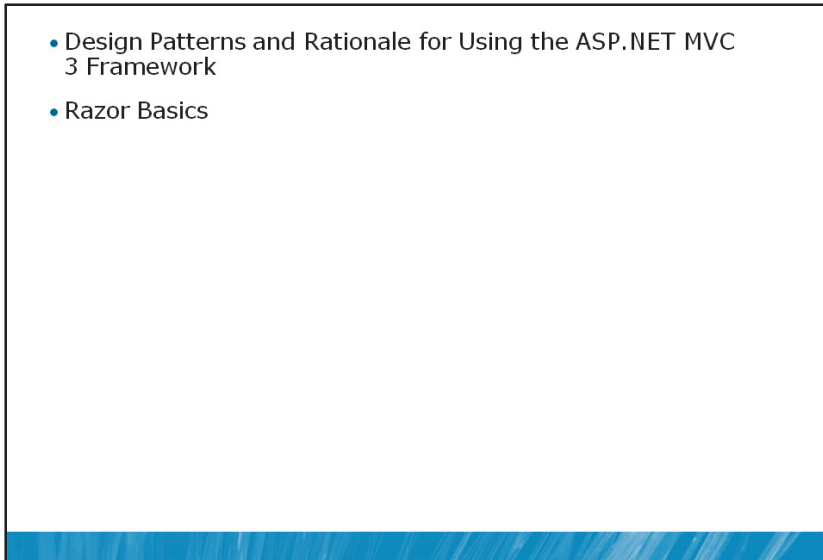


FIGURE 9.1: SITE REPORT

With the features previously mentioned, WebMatrix provides an easy yet powerful set of tools for to create new websites and manage existing websites.

Lesson 2

ASP.NET MVC 3 and Razor



In this lesson, you are introduced to ASP.NET MVC 3, Razor view engine basics, and compatible Visual Studio 2010 templates used to create and manage ASP.NET MVC 3 websites. ASP.NET MVC 3 is based on the foundations of ASP.NET and adds an implementation of the popular MVC design pattern.

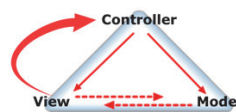
You will also be presented with the design pattern and the rationale for using the ASP.NET MVC 3 framework and supported templates provided by Visual Studio 2010. In addition, you will be introduced to a new view engine used in ASP.NET MVC 3 applications, called *Razor*. The Razor view engine is also supported by WebMatrix and can be used with the options you learned in lesson 1.

Lesson Objectives

After completing this lesson, you will be able to use ASP.NET MVC 3 and the Razor view engine as the main frameworks for building web applications.

Design Patterns and Rationale for Using the ASP.NET MVC 3 Framework

- MVC is a patterns-based framework which enables clean separation of concerns by dividing the application into different loosely coupled parts
- MVC stands for Model-View-Controller
- Model is application's logic related to data
 - Usually responsible for retrieving and storing data
- View is application's UI layer responsible for presenting the data from model
- Controller is application's interaction handler, also responsible for selecting and matching between model and view



What is ASP.NET MVC?

ASP.NET MVC is a powerful, patterns-based framework for building dynamic websites. It is designed to encourage a clean *separation of concerns*, so the code used to create a website is split into small modules or layers that make management and maintenance relatively easy. ASP.NET MVC gives you, the developer, full control over markup and enables agile development.

What is MVC?

MVC stands for *Model-View-Controller*, an architectural pattern that separates an application into three main components: the model, the view, and the controller. The ASP.NET MVC framework provides an alternative to the ASP.NET Web Forms framework for creating MVC-based web applications.



Additional Reading: ASP.NET Web Forms are pages that users request through their browser and that form the user interface (UI) that gives web applications their look and feel. These pages are written by using a combination of HTML, server controls, and server code. When users request a page, it is compiled and executed on the server, and then it generates the HTML markup that the browser can render.

You can read more about ASP.NET Web Forms at the following location:

[ASP.NET Webforms](#)

As already described, the MVC framework includes three main components:

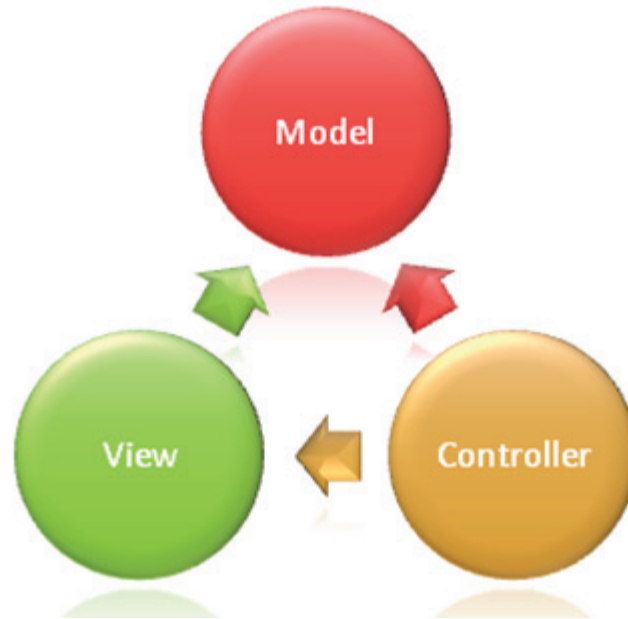


FIGURE 9.2: MVC COMPONENTS

Model—The parts of the application that implement the logic related to the application data. Usually, the model is responsible for retrieving and storing data in the database. Note that in some cases, the model is only a logical separation, rather than a physical separation. Models classes are put in the Models folder in the MVC project.



Note: A model can be any class that you create.

View—The parts of the application that implement and display the application user interface (UI). In most cases, the UI is created from data provided by the model. Views are put in the Views folder in the MVC project.

The following example shows how to use the view component.

View

```

<!DOCTYPE html>
<html>
<head>
<meta charset="utf-8" />
<title>@ViewBag.Title</title>
</head>
<body>
<div class="page">
<header>
<div id="title">
<h1>My MVC Application</h1>
</div>
<nav>
<ul id="menu">
<li>@Html.ActionLink("Home", "Index", "Home")</li>
<li>@Html.ActionLink("About", "About", "Home")</li>
</ul>
</nav>
</header>

```

```
<section id="main">
</section>
<footer>
</footer>
</div>
</body>
</html>
```

Controller—The parts of the application that handle the application user interaction, work with models and are responsible for selecting a view. The controller is solely responsible for selecting the view to suit the model data, and it handles all the interactions required by view and model. Controllers are put in the Controllers folder in the MVC project.

The following example shows how to use a simple controller.

Controller

```
public class HomeController : Controller
{
    public ActionResult Index()
    {
        ViewBag.Message = "Welcome to ASP.NET MVC!";
        return View();
    }
    public ActionResult About()
    {
        return View();
    }
}
```

This way, the MVC pattern helps create applications that separate the different aspects of the application (input logic, business logic, and UI logic), while providing loose coupling between these elements.



Additional Reading: A loosely coupled system is a system whose components are minimally dependent on other components in the system. This usually makes it easy to modify or replace components in a predictable, isolated manner without affecting the rest of the system.

The MVC pattern specifies where each kind of logic should be located in the application.

- The UI logic belongs in the view.
- Input logic belongs in the controller.
- Business logic belongs in the model.

This separation helps developers manage complexity when building an application, because developers can focus on one aspect of the implementation at a time.

The ASP.NET MVC Framework implements the MVC pattern that provides the following features:

- Clean separation of application tasks, testability, and test-driven development
- Extensible and pluggable framework, which enables easy customization and replacement
- Powerful URL-mapping mechanism, which enables searchable URLs within the application

Because it is built on top of ASP.NET, the ASP.NET MVC also supports all of the following ASP.NET features:

- Re-use of existing markup from ASP.NET pages (.aspx), ASP.NET controls (.ascx) and master pages (.master) as view templates
- Forms and Windows authentication, URL authorization, membership and roles, data caching, and others
- Inline ASP.NET expressions (<%= ... %>), server-side controls, data-binding, localization, and so on

When to Use ASP.NET MVC?

ASP.NET MVC is a server-side framework that is meant to be used as the backend technology for websites and applications. It is developed using Visual Studio and built-in ASP.NET MVC project templates unlike WebMatrix which is an IDE by itself. When building complex websites and applications you will probably want to use ASP.NET MVC as your framework and not tools like WebMatrix. The main reasons for that are:

- The separation of concerns that were discussed earlier in this topic.
- The testability of ASP.NET MVC which helps to create Unit Tests for most of component you will build.
- The extensibility points in the framework

Razor Basics

- Razor is a new view engine used in ASP.NET MVC projects
- Introduces "@" as a code block starter
- Can distinguish markup commands and static content (such as email addresses)
- Enables basic programming language constructs as part of static HTML markup
- Supports if-blocks, loops, and built-in and custom helper functions
- Great IntelliSense support provided by Visual Studio
- Can still use simple text editors (like Notepad) to write Razor code
- Easy to learn and use

```
<p>
Welcome to MOC #@courseID, chapter @chapterID!
</p>
```

What is Razor?

ASP.NET MVC introduces a new view engine, called *Razor*. Razor is optimized around HTML and HTML5 generation and uses a code-focused template approach. Razor is a modern view engine and designed to provide an easy coding solution with the following characteristics:

- Compact and expressive
- Easy to learn and use
- Does not introduce any new programming language
- Does not require a special text editor, but has great IntelliSense capabilities within Visual Studio
- Enables unit testing

When writing views in Razor, you will use HTML5 as the building block for the view and you will use the Razor syntax to help you create the views using ASP.NET MVC.

Getting Started with Razor

Visual Studio 2010 provides native support for ASP.NET MVC projects and enables you to create views marked up by using the Razor view engine.

To start an ASP.NET MVC 3 project, select the **ASP.NET MVC 3 Web Application** project template when you create a new project. While creating a new project or adding a new View to an existing project, Visual Studio enables you to select the Razor view engine for the view.

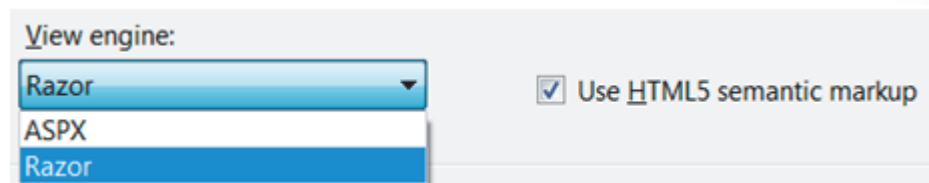


FIGURE 9.3: SELECTING THE VIEW ENGINE FOR THE VIEW

Razor introduces a new modified syntax that is different from the ASP.NET inline expression syntax. The Razor syntax uses the "@" sign to start a new code block. It enables you to start with static HTML content and then add dynamic markup.

The following example shows how to mix static HTML and dynamic server code.

Razor sample

```
<p>
Welcome to MOC #@courseID, chapter @chapterID!
</p>
```



Note: This code example uses two variables, (**courseID** and **chapterID**), defined elsewhere in the code. The definition is omitted for brevity.

Razor syntax enables easy looping through data provided by model, extensive if-blocks, and multiline statements.

The following example shows a code sample that introduces the if-block.

If-block with Razor

```
@if (chapterID != 9)
{
<p>This is not a "Razor" chapter...</p>
}
else
{
<p>This is a "Razor" chapter!</p>
}
```



Note: This code example uses a variable (**chapterID**) defined elsewhere in the code. The definition is omitted for brevity.

Razor also enables built-in .NET functionality (for example, the **DateTime** class) and ASP.NET MVC specific HTML helpers, which are code blocks that encapsulate generating HTML markup, as part of the output markup.

```
<p>This page was generated at @DateTime.Now</p>
<div>
    @Html.TextBox("courseID")
</div>
```

This code block causes the Razor view engine to generate the following markup.

```
<p>This page was generated at 12/16/2011 9:51:26 AM</p>
<div>
    <input id="courseID" name="courseID" type="text" value="" />
</div>
```



Note: The generated code can be reviewed by using the Microsoft Internet Explorer® built-in source viewer. To view the code, right-click the webpage, and then click **View Source**.

Razor also enables defining custom helpers using the **@helper** syntax.

```
@helper Echo(string output){
    <h1>@output</h1><br/>
}
<div>@Echo("Hello HTML5 MOC!")</div>
```



Note: Custom helpers can help to re-use code blocks and functionality.

Razor enables you to re-use markup and organize the views by creating *layouts*. Layouts are CSHTML files that hold common content for multiple pages and enable you to create specific pages focused on the content. The common parts of the webpage are developed once and stored in a layout file. Layout files are stored in the **\Views\Shared** folder. The layout file is referenced from the view file.

To reference the layout file from the view, set the **Layout** property as shown in the following code.

```
@{
    Layout = "~/Views/Shared/CommonLayout.cshtml";
    ViewBag.Title = "Students Report";
}
```



Note: This code references the **CommonLayout.cshtml** file created and stored in the **\Views\Shared** project folder. The tilde (~) is used to indicate that the URL used should start from the virtual directory root of the website.

As already described, the layout files hold common content for the pages, but at some locations must generate view-specific content. This is achieved by using the **@RenderBody** and **@RenderSection** commands.

- The **@RenderBody** command generates the view content, in place of the command.

```
...
<div>
    @RenderBody()
</div>
...
```

- The **@RenderSection** command generates markup for a specific part of the page in place of the command.

```
@RenderSection("CustomSection")
```



Best Practice: Before generating the section, check if it is defined in the view by using the **IsSectionDefined** function as shown in the following example.

```
@if (IsSectionDefined("CustomSection"))
{
    @RenderSection("CustomSection")
}
```



Note: Please note that **@RenderBody** can be used only once in the layout file. Also, **@RenderSection** for a specific section can be used only once in the entire layout file. However, it can be used multiple times for different sections.

Following from the **@RenderSection** description, the section definition comes in a specific view file. The section definition can contain any valid Razor syntax, including static content and any Razor commands. It can be placed anywhere in the view file and can be defined as shown in the following example.

```
@section CustomSection {
    <h1>This is a custom generated section</h1>
}
```

In addition to enabling layout management, Razor enables you to associate the view implementation with a specific model. The model is defined in the view by using the **@model** keyword.

```
@model Dictionary<string, List<MOCMVCQuickStart.Models.Game>>
```



Note: The preceding code defines a dictionary of Xbox games, sorted by genre, and will be used in the Lab associated with this lesson.

This not only connects the view with its model (and the data in the model), but also enables IntelliSense in the Visual Studio 2010 editor, which helps to develop the view more efficiently by using strongly-typed known types.



FIGURE 9.4: INTELLISENSE SUGGESTIONS USING A STRONGLY-TYPED MODEL

Razor also supports commenting out parts of the view by using **@*...*@** syntax. Anything between the opening **@*** and closing ***@** symbols is commented out.

```
@*
This code is commented out.
@foreach (var grade in Model)
{
    <h2 class="header">@grade.Key</h2>
}
*@
```

All these features make Razor a great new view engine option that is streamlined for code-focused template usage, with compact syntax that greatly simplifies development and improves code readability.

Lab: Using WebMatrix and Other Development Tools

- Using WebMatrix to Create a Website
- Using ASP.NET MVC 3 to Create a Website

Logon Information

Virtual Machine	Virtual machine: SEA-DEV
User Name	User name: Admin
Password	Password: Pa\$\$w0rd

Estimated Time: 40 minutes

Scenario

The Contoso, Ltd. management has asked you to create two Proofs of Concept (POC) for building the online store, one with WebMatrix and the other with ASP.NET MVC 3. After you create these POCs, the management decide the platform on which to continue and create the backend for the web application.

Objectives

After completing this lab, you will be able to:

- Build the home page by using WebMatrix.
- Build the home page by using ASP.NET MVC 3.

Exercise 1: Using WebMatrix to Create a Website

Scenario

Create a WebMatrix POC project to reproduce a static HTML site, similar to the Module 1 lab, by using WebMatrix.

► Task 1: Create a new website.

1. Create a new POC site.

► Task 2: Add existing items to the project.

1. Add images to the project.
2. Add JavaScript script files to the project.
3. Add CSS files to the project.
4. Add the **web.config** file to the project.

► Task 3: Add a new HTML file to the project and edit its content.

1. Add the **Homepage.html** file to the project.
2. Change the title of the **Homepage.html** page.
3. Add CSS and JavaScript file references to the **Homepage.html** page.
4. Add content to the **Homepage.html** page.

► Task 4: Run the website.

1. Run the POC website.

Results: A WebMatrix POC project is created with the layout of the first module's lab solution.

Exercise 2: Using ASP.NET MVC 3 to Create a Website

Scenario

Use Visual Studio 2010 and the ASP.NET MVC 3 Project Template to reproduce a static HTML site, similar to the Module 1 lab.

► **Task 1: Create and configure a new ASP.NET MVC 3 project.**

1. Create a new ASP.NET MVC 3 web application project.
2. Configure the new project.

► **Task 2: Add existing assets and modify the CSS and CSHTML files.**

1. Add the global and home page script files to the project.
2. Add home page images to the project.
3. Add CSS styles to the project.
4. Modify `_Layout.cshtml`.

► **Task 3: Create a model.**

1. Add a new model class and context.

► **Task 4: Create a controller.**

1. Add a new controller.

► **Task 5: Create a view.**

1. Add a new view.

► **Task 6: Run the website.**

1. Run the project.

Results: An ASP.NET MVC 3 POC project is created with the layout of the first module's lab solution.

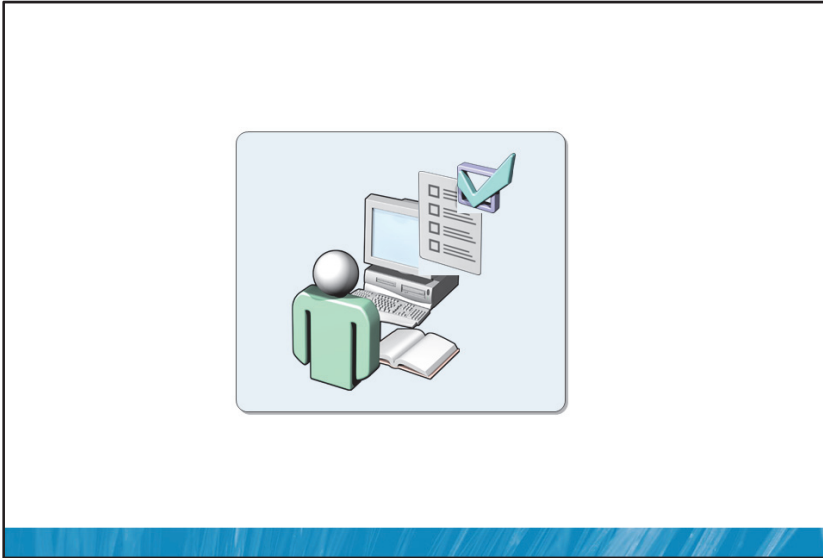
Module Review and Takeaways

Microsoft Web Platform provides a rich set of tools to help develop modern websites by using HTML5. The tools include WebMatrix and Visual Studio 2010. In Visual Studio 2010, you can develop websites by using ASP.NET MVC 3, which enables you to use the new Razor view engine. ASP.NET MVC 3 greatly simplifies the development process and provides native support for a modern view-oriented framework—MVC.

Tools

Tool	Use for	Where to find it
WebMatrix; Visual Studio 2010 with ASP.NET MVC 3 support	Creating, customizing, and publishing websites.	http://go.microsoft.com/fwlink/?LinkID=145505

Course Evaluation



Keep this evaluation topic page if this is the final module in this course. Insert the Product_Evaluation.ppt on this page.

If this is not the final module in the course, delete this page

Your evaluation of this course will help Microsoft understand the quality of your learning experience.

Please work with your training provider to access the course evaluation form.

Microsoft will keep your answers to this survey private and confidential and will use your responses to improve your future learning experience. Your open and honest feedback is valuable and appreciated.

Module 1: Introduction to HTML5 Development

Lab: Building a Basic Webpage in the Visual Studio 2010 SP1 Environment

Exercise 1: Writing the Home Page HTML

► Task 1: Create a new Empty Web Application.

1. Log on to the virtual machine, **SEA-DEV**, as **Admin**, with the password, **Pa\$\$w0rd**.
2. Open Microsoft® Visual Studio® 2010.
3. Open the solution, **GameStore.sln**, from the **D:\Mod01\Labfiles\Starter** folder.
4. In the **GameStore** solution, right-click the **GameStore** solution in Solution Explorer.
5. In the **Add** menu, click **New Project**.
6. In the **Add New Project** dialog box, click **Visual C#** under **Installed Templates**, and then click **Web**.
7. On the web tab, select the **ASP.NET Empty Web Application**.
8. Set the name of the web application to **GameStore.Html**.
9. Click **OK** to create the web application.

► Task 2: Create the solution structure.

1. In the **GameStore.Html** project, right-click the **GameStore.Html** project in Solution Explorer.
2. In the **Add** menu, click **New Folder**.
3. Name the created folder, **Images**.
4. In the **GameStore.Html** project, right-click the **Images** folder.
5. In the **Add** menu, click **Existing Item**.
6. In the **Add Existing Item** dialog box, go to the **D:\Mod01\Labfiles\Starter \Images** folder.
7. Choose all the images in the folder (using Ctrl+A).
8. Click **Add** to add all the images.

► Task 3: Create the Homepage HTML file.

1. In the **GameStore.Html** project, right-click the project in the Solution Explorer.
2. In the **Add** menu, click **New Item**.
3. In the **Add New Item** dialog box, click **HTML Page**.
4. In the Name text box, call the page, **homepage.html**, and then click **Add**.
5. Right-click the **homepage.html**.
6. Click the **Set As Start Page** menu item to set the homepage as the start page for the web application.
7. In the created file, replace the existing code with the code that exists in the **homepageStructure.txt** file, which is located in the **D:\Mod01\Labfiles\Starter\Assets** folder.



Note: The structure that exists in the **homepageStructure.txt** file is meant to be the basic for building the HTML webpage. You should explore the structure to get a feeling understanding of how to build a whole webpage.

► **Task 4: Create a simple HTML element.**

1. In the **homepage.html**, locate the div with the **catalog** id.
2. Inside the div, find the HTML comment: `<!-- Genres -->`. Instead of this comment, you will implement the genres and game details HTML elements.
3. Replace the comment with a genre div.

```
<div class="genre"></div>
```

4. Inside the genre div, add a header div with the following heading.

```
<div class="genre">
  <div class="header">
    <h2>Strategy Games</h2>
  </div>
</div>
```

5. Inside the genre div and below the header div, add a games div.

```
<div class="genre">
  <div class="header">
    <h2>Strategy Games</h2>
  </div>
  <div class="games">
  </div>
</div>
```

6. Inside the games div, add a game div that includes the game image, the game title, and a link.

```
<div class="genre">
  <div class="header">
    <h2>Strategy Games</h2>
  </div>
  <div class="games">
    <div class="game">
      <p>
        
      </p>
      <h3 class="game-text">Age of Empires</h3>
      <p class="money-text">$30</p>
      <div class="button add-to-cart-button">
        Add to Cart
      </div>
      <a class="learn-more" href="#">Learn More ></a>
    </div>
  </div>
</div>
```

7. Save the file by pressing Ctrl+S.
8. Run the web application by pressing Ctrl+F5 and view the homepage you created.

► **Task 5: Create a simple HTML table element.**

1. In the **homepage.html**, locate the div with the `<!-- Cart Table -->` comment.
2. Instead of the comment, create a new HTML table.

```
<div class="content">
<table>
  </table>
</div>
```

3. Create a table row to hold the chosen game.

```
<div class="content">
<table>
  <tr>
  </tr>
</table>
</div>
```

4. Inside the table row, create three table cells.

```
<div class="content">
<table>
  <tr>
    <td>
    </td>
    <td>
    </td>
    <td>
    </td>
  </tr>
</table>
</div>
```

5. Populate the cells with the game title, game image, and the X image.

```
<div class="content">
<table>
  <tr>
    <td>
      
    </td>
    <td>Kinect Star Wars</td>
    <td>
      
    </td>
  </tr>
</table>
</div>
```

6. Save the file by pressing Ctrl+S.
7. Run the web application by pressing F5 and view the homepage you created.

► **Task 6: Create a simple unordered list.**

1. In the **homepage.html**, locate the div with the nav class which holds a `<!-- unordered list -->` comment.
2. Instead of the comment create a new unordered list:

```
<div class="nav">
<ul>
  </ul>
</div>
```

3. Inside the unordered list, create two list items.

```
<div class="nav">
<ul>
  <li></li>
  <li></li>
</ul>
</div>
```

4. Populate the list items with hyperlinks.

```
<div class="nav">
<ul>
  <li><a class="page-link active" href="Homepage.html">Games</a></li>
  <li><a class="page-link" href="#">Checkout</a></li>
</ul>
</div>
```

5. Save the file by pressing Ctrl+S.
6. Run the web application by pressing F5 and view the homepage you created.

Results: A home page written in HTML containing a game catalog, a game in spotlight, and cart data.

Exercise 2: Styling the Homepage

► Task 1: Create the basic theme for the homepage.

1. In the **GameStore.Html** project, right-click the project in the Solution Explorer.
2. In the **Add** menu, click **New Folder**.
3. Name the created folder, **Styles**.
4. In the **GameStore.Html** project, right-click the **Styles** folder.
5. In the **Add** menu, click **New Item**.
6. In the **Add New Item** dialog box, choose **Style Sheet**.
7. In the **name** text box, call the style sheet, **theme.css**.
8. Click the **Add** button to add the new file.
9. Open the **homepage.html** file.
10. In the head section, add the link to the created style sheet.

```
<head>
...
<link href="Styles/theme.css" rel="stylesheet" type="text/css" />
</head>
```

11. Save the file by pressing Ctrl+S.
12. Open the **theme.css** file.
13. Add the default styles for elements.

```
body
{
    font-family: Arial, Helvetica, sans-serif;
    text-align: center;
}
a
{
    color: white;
    text-decoration: none;
}
h1, h2, h3
{
    padding: 0px;
    margin: 0px;
    font-weight: bold;
    font-size: 1em;
}
```

14. Save the file by pressing Ctrl+S.
15. Add the header class name, which gives a white color and bold font weight to the element by using the class.

```
.header
{
    color: white;
    font-weight: bold;
}
```

16. Add the button class name, which styles all the buttons by using the class name.

```
.button
{
    background-image: url(../images/plain-button.png);
    text-align: center;
    float: left;
    height: 28px;
    line-height: 28px;
    background-repeat: no-repeat;
    color: white;
    font-weight: bold;
    font-size: 16px;
    vertical-align: middle;
    margin: 0px auto;
    cursor: pointer;
    width: 126px;
}
```

17. Add the text class name, which defines the style for texts that are displayed in the application.

```
.text
{
    color: #5b5b5b;
    font-weight: bold;
    font-size: 12px;
    text-align: left;
    line-height: 15px;
}
```

18. Add the container class name, which defines how to display the main div in the application.

```
.container
{
    width: 970px;
    height: 100%;
    margin: 0px auto;
}
```

19. Save the file by pressing Ctrl+S.

20. Add the h1 override.

```
h1
{
    text-align: left;
    font-size: 32px;
    color: #5c5c5c;
    padding: 24px 0px 10px;
}
```

21. Add the logo layout by using the id selector.

```
#logo
{
    font-size: 48px;
    font-weight: bold;
    font-variant: small-caps;
    color: #2a801f;
    text-align: left;
    padding: 0px 5px;
    margin: 10px 0px;
}
```

```

    text-shadow: 2px 2px 2px #000;
}

```

22. Add the navigation bar styles.

```

.nav
{
    height: 40px;
    width: 100%;
    background-color: #e5e5e5;
    margin-bottom: 5px;
}
.nav ul
{
    list-style: none;
    padding: 0px;
}
.nav ul li
{
    display: block;
}

```

23. Add the footer styles.

```

.container > .footer
{
    clear: both;
    width: 100%;
    background-color: #e5e5e5;
    margin: 50px auto 10px;
    float: left;
}
.container > .footer p
{
    padding: 5px 20px;
    float: left;
}

```

24. Add the following CSS styles to the file.

```

h2
{
    font-size: 18px;
}
table
{
    border-collapse: collapse;
    border-spacing: 0px;
}

.button-bar
{
    clear: both;
    text-align: left;
    padding: 10px;
}

.button-bar > .button
{
    margin: 0px 2px;
}

```

```

}

.info-icon
{
    width: 35px;
    height: 35px;
}
.box > .header
{
    height: 38px;
    line-height: 38px;
    background-color: #949494;
}
.box.narrow
{
    padding-top: 20px;
    width: 177px;
}
.xbox-link
{
    color: #2a801f;
    text-decoration: underline;
    padding-left: 3px;
}
.page-link
{
    height: 40px;
    width: 180px;
    font-size: 19px;
    font-weight: bold;
    float: left;
    padding-top: 9px;
    background-repeat: no-repeat;
    background-image: url(../images/page-button.png);
}
.page-link.active
{
    background-image: url(../images/page-button-active.png);
}
.money-text
{
    font-size: 18px;
    color: #2a801f;
    font-weight: bold;
    line-height: 15px;
    padding-top: 15px;
}

```

25. Save the file by pressing Ctrl+S.

► **Task 2: Lay out the main homepage elements.**

1. In the **GameStore.Html** project, right-click the **Styles** folder.
2. In the **Add** menu, click **Existing Item**.
3. In the **Add Existing Item** dialog box, choose all the CSS files in the **D:\Mod01\Labfiles\Starter\Css** folder.
4. Click **Add**.
5. Open the **homepage.html** file.
6. In the head section, add the link to the created style sheet.

```
<head>
...
<link href="Styles/theme.css" rel="stylesheet" type="text/css" />
<link href="Styles/slideshow.css" rel="stylesheet" type="text/css" />
<link href="Styles/spotlight.css" rel="stylesheet" type="text/css" />
<link href="Styles/catalog.css" rel="stylesheet" type="text/css" />
<link href="Styles/cart.css" rel="stylesheet" type="text/css" />
<link href="Styles/dragdrop.css" rel="stylesheet" type="text/css" />
</head>
```

7. Save the file by pressing Ctrl+S.
8. Run the application by pressing F5.



Note: The homepage will include styling and layout. You should check the implementation inside all the supplied CSS files to understand how the styling of elements like the cart was created.

Results: A homepage with a layout and style in accordance with the UI mockup image document.

Exercise 3: Adding JavaScript and jQuery to Make a Dynamic Home Page

► Task 1: Create a new JavaScript file.

1. In the **GameStore.Html** project, right-click the project.
2. In the **Add** menu, click **New Folder**.
3. Name the created folder **Scripts**.
4. In the **GameStore.Html** project, right-click the **Scripts** folder.
5. In the **Add** menu, click **New Item**.
6. In the **Add New Item** dialog box, under the web tab, click the Jscript file.
7. In the **name** text box, call the JavaScript file, **GlobalFunctions.js**.
8. Click the **Add** button to add the new file.
9. Open the **homepage.html** and in the head section, add the following script reference.

```
<head>
...
<script src="Scripts/GlobalFunctions.js" type="text/javascript"></script>
</head>
```

10. Save the file by pressing Ctrl+S.

► Task 2: Add jQuery files and jQuery IntelliSense.

1. In the **GameStore.Html** project, right-click the **Scripts** folder.
2. In the **Add** menu, click **Existing Item**.
3. In the **Add Existing Item** dialog box, go to the **D:\Mod01\Labfiles\Starter\Scripts** folder.
4. Choose both jQuery files, **jquery-1.5.1.js** and **jquery-1.5.1-vsdoc.js**, in the folder (using Ctrl+A).
5. Click the **Add** button to add all the script files.
6. Open the **GlobalFunctions.js** file. Add the following line at the head of the file to enable jQuery IntelliSense.

```
/// <reference path="jquery-1.5.1-vsdoc.js" />
```

7. Open **homepage.html** and in the head section, add the following script reference.

```
<head>
...
<script src="Scripts/jquery-1.5.1.js" type="text/javascript"></script>
<script src="Scripts/GlobalFunctions.js" type="text/javascript"></script>
</head>
```

8. Save the file by pressing Ctrl+S.

► Task 3: Create simple JavaScript objects.

1. In the **GlobalFunctions.js** file, create a Cart object.

```
var Cart = (function () {
})();
```

2. Add a games field to the Cart object, which will hold an array.

```
var Cart = (function () {
    this.games = new Array();
})();
```

Pay attention to use **this** to create the field.

3. Create a Game object.

```
var Game = (function () {
})();
```

4. Add three fields (gameId, name, and imageUrl) to the game object.

```
var Game = (function () {
    this.gameId = 0;
    this.name = "";
    this.imageUrl = "";
})();
```

Pay attention to the default values that every instance of the Game object will have when it is created.

5. Save the file by pressing Ctrl+S.

► Task 4: Add event listeners to the JavaScript file.

1. Add the following function to the file.

```
function contentLoaded() {
}
```

2. Wire the **DOMContentLoaded** event to the **contentLoaded** function created in step 1 by using the following code, which should be written at the end of the file *outside* the **contentLoaded** function.

```
document.addEventListener("DOMContentLoaded", contentLoaded, false);
```

3. Save the file by pressing Ctrl+S.

► Task 5: Create the collapsible panel functionality.

1. Create the addCollapsiblePanels function that will be used to create the collapsible panels.

```
function addCollapsiblePanels() {
}
```

2. Use jQuery class selector to select all the genres.

```
function addCollapsiblePanels() {
    $(".genre")
}
```

3. Use the jQuery each function to iterate all the selected genres.

```
function addCollapsiblePanels() {
    $(".genre").each(function () {
    });
}
```

4. Create an element variable and set it to reference the current genre div.

```
function addCollapsiblePanels() {  
    $(".genre").each(function () {  
        var element = $(this);  
    });  
}
```

5. For the element, find its header by using the jQuery selector and wire a click event to it.

```
function addCollapsiblePanels() {  
    $(".genre").each(function () {  
        var element = $(this);  
        element.find("> .header").click(function () {  
        });  
    });  
}
```

6. In the click event, find the div with the games class name and toggle it.

```
function addCollapsiblePanels() {  
    $(".genre").each(function () {  
        var element = $(this);  
        element.find("> .header").click(function () {  
            element.find(".games").toggle();  
        });  
    });  
}
```

7. Create the collapseAllPanels function that will be used to collapse all the games div under the previous created function.

```
function collapseAllPanels() {  
}
```

8. Use the jQuery selector to select all the elements with the games class name inside the collapseAllPanels function.

```
function collapseAllPanels() {  
    $(".games");  
}
```

9. Toggle all the selected elements.

```
function collapseAllPanels() {  
    $('.games') .toggle();  
}
```

10. Add the call to the functions in the contentLoaded function.

```
function contentLoaded() {  
    addCollapsiblePanels();  
    collapseAllPanels();  
}
```

11. Save the file by pressing Ctrl+S.

► Task 6: Create the slideshow functionality.

1. Create a new function called `switchSlide` that will be used to switch slides in the slideshow.

```
function switchSlide() {
}
```

2. Implement the `switchSlide` function, which switches the current slide to the next slide .

```
function switchSlide() {
    // get the current active image in the slideshow
    var active = $("#slideshow img.active");
    // if there is no active image, search for the last image in the slideshow
    // (this will select the current image when the page first loads)
    if (active.length == 0) {
        active = $("#slideshow img:last");
    }
    // pick the next image to show in the slideshow
    var next = active.next().length ? active.next() : $("#slideshow img:first");
    // add the "last-active" style to the current active image to give it a smaller z-index
    active.addClass("last-active");
    // animate the slide fade out and set the next image as the current active image
    next.css({ opacity: 0.0 })
        .addClass("active")
        .animate({ opacity: 1.0 }, 1000, function () {
            active.removeClass("active last-active");
        });
}
```

3. Create a new function called, `startSlideShow`, which will start the slideshow and switch between slides every eight seconds or if the slideshow image is clicked.

```
function startSlideShow() {
    // set the interval to switch between images every 8 seconds
    setInterval("switchSlide()", 8000);
    // switch slides whenever the slideshow container is clicked
    $("#slideshow").click(function () {
        switchSlide();
    });
}
```

4. Add the call for the function in the `contentLoaded` function.

```
function contentLoaded() {
    startSlideShow();
    addCollapsiblePanels();
    collapseAllPanels();
}
```

5. Save the file by pressing `Ctrl+S`.
6. Run the web application by pressing `F5`.

Results: A home page containing dynamic parts such as collapsible panels and an image slideshow, manipulated by using JavaScript and jQuery, as described in the design document

MCT USE ONLY. STUDENT USE PROHIBITED

Module 2: Creating Page Structure and Navigation

Lab: Creating Page Structure and Navigation

Exercise 1: Using Semantic Elements in the Home Page

► **Task 1: Replace the existing HTML elements with the new HTML5 elements.**

1. Log on to the virtual machine named **SEA-DEV** as **Admin**, with the password, **Pa\$\$w0rd**.
2. Open Microsoft® Visual Studio® 2010.
3. Open the solution, **GameStore.sln**, from the **D:\Mod02\Labfiles\Starter** folder.
4. Open the **homepage.html** file by double-clicking it.
5. Add a header element to wrap the logo and navigation elements.

```
<header>
<p id="logo">Contoso, Ltd</p>
<div class="nav">
<ul>
<li><a class="page-link active" href="Homepage.html">Games</a></li>
<li><a class="page-link" href="#">Checkout</a></li>
</ul>
<a class="go-to-cart-link money-text" href="#">Go to Cart </a>
</div>
</header>
```

6. Replace the **div** element with a class attribute **nav** with a **nav** element.

```
<header>
<p id="logo">Contoso, Ltd</p>
<nav>
<ul>
<li><a class="page-link active" href="Homepage.html">Games</a></li>
<li><a class="page-link" href="#">Checkout</a></li>
</ul>
<a class="go-to-cart-link money-text" href="#">Go to Cart </a>
</nav>
</header>
```

7. Replace the **div** element with id attribute **slideshow** with a **section** element, also with id attribute of **slideshow**.

```
<section id="slideshow">



</section>
```

8. Change all the **divs** in the **spotlight** section to **headers**, **sections**, and **footers**, and change the main **div** of the spotlight to an **aside**.

```
<aside id="spotlight" class="box">
<header>
    Spotlight
```

```

</header>
<section class="content">
</section>
<footer>
    Amazing Entertainment
</footer>
</aside>

```

9. Change the **cart** element to an **aside** element with **header**, **section**, and **footer** children. In addition, make the **div** element with the drag class name, a **section**.

```

<aside id="cart" class="box narrow">
<header>
    Games in Cart
</header>
<section class="content">
<table>
<tr>
<td>

</td>
<td class="text">Kinect Star Wars</td>
<td>

</td>
</tr>
<tr>
<td>

</td>
<td class="text">Kinect Star Wars</td>
<td>

</td>
</tr>
<tr>
<td>

</td>
<td class="text">Kinect Star Wars</td>
<td>

</td>
</tr>
</table>
</section>
<footer>
<a href="Cart.html">
<div class="cart-icon"></div>
</a>
</footer>
<section class="drag">
<div class="drag-icon">

</div>
<div class="text">
    You can drag items into the cart
</div>
</section>
</aside>

```

10. Change the **div** with **catalog** id attribute into a **section**.

```
<section id="catalog">
</section>
```

11. Change the elements with class name **genre** into **sections** and their **headerdivs** into **header** elements.

```
<section class="genre">
<header>
<h2>Strategy Games</h2>
</header>
</section>
```

12. Change every game div into an article element. Change the first paragraph element into figure element and the second paragraph to be a header element. Also change the game-id attribute into data-game-id with a new id for each attribute.

```
<article class="game" data-game-id="1">
<figure>

</figure>
<header>
<h3 class="game-text">Age of Empires</h3>
</header>
<p class="money-text">$30</p>
<div class="button add-to-cart-button">
    Add to Cart
</div>
<a class="learn-more" href="GameDetails.html">Learn More ></a>
</article>
```

13. Change the last **div**, with the **footer** class attribute, into a **footer** element.

```
<footer>
<p class="text">Copyright <a class="xbox-link" href="http://www.xbox.com">Xbox Site</a></p>
</footer>
```

14. Save the file by pressing Ctrl+S.
15. Run the application by pressing Ctrl+F5, and see the results.



Note: The styles that affect the layout of the webpage are no longer valid for the new HTML elements. Changing the HTML without changing the styles that affect its layout will result in breaking changes to the displayed webpage. In the next task, we will fix it.

► Task 2: Update the style to reflect the HTML changes.

1. Open the **theme.css** file by double-clicking it.
2. Change the **.header** selector into a **header** selector.

```
header
{
    color: white;
    font-weight: bold;
}
```

and


```
.box > header
{
    height: 38px;
    line-height: 38px;
    background-color: #949494;
}
```

3. Change the **.footer** selector into a **footer** selector.

```
.container > footer
{
    clear: both;
    width: 100%;
    background-color: #e5e5e5;
    margin: 50px auto 10px;
    float: left;
}
.container > footer p
{
    padding: 5px 20px;
    float: left;
}
```

4. Change the **.nav** selector into a **nav** selector.

```
nav
{
    height: 40px;
    width: 100%;
    background-color: #e5e5e5;
    margin-bottom: 5px;
}
nav ul
{
    list-style: none;
    padding: 0px;
}
nav ul li
{
    display: block;
}
```

5. Save the file by pressing Ctrl+S.
6. Open the **spotlight.css** file by double-clicking it.
7. Change the **.footer** selector into **footer** selector.

```
#spotlight > footer
{
    background: #2a6da9;
    height: auto;
    color: white;
    font-size: 15px;
    font-weight: bold;
}
```

8. Save the file by pressing Ctrl+S.
9. Open the **catalog.css** file by double-clicking it.
10. Change the **.header** selector into a **header** selector.

```
.genre > header
{
    height: 35px;
    line-height: 35px;
    background-image: url(../images/strip.png);
    background-repeat: no-repeat;
    cursor: pointer;
}
```

11. Save the file by pressing Ctrl+S.
12. Open the **cart.css** file by double-clicking it.
13. Copy the following code, replacing the content of the file.

```
#cart
{
    height: 100%;
}
aside#cart
{
    float: right;
}
#cart > .content
{
    height: 100%;
    color: white;
    font-size: 19px;
    font-weight: bold;
}
#cart > footer
{
    width: auto;
    height: 38px;
    color: white;
    font-size: 15px;
    font-weight: bold;
    background-color: #949494;
    position: relative;
}
#cart table
{
    width: 100%;
    height: 100%;
    background-color: #fbfbfb;
    border-spacing: 0px;
}
.cart-icon
{
    position: absolute;
    top: 0px;
    right: 0px;
    width: 25px;
    height: 25px;
    background-image: url(../images/cart-icon.png);
    background-repeat: no-repeat;
    margin: 5px;
}
#cart .thumbnail
{
    width: 48px;
    height: 65px;
}
```

```

.cart-remove-icon
{
    width: 19px;
    height: 19px;
}
#cart .text
{
    padding: 0px 4px;
}
.add-to-cart-button
{
    background-image: url(../images/cart-button.png);
    text-indent: 26px;
    float: none;
}
.go-to-cart-link
{
    width: 130px;
    background-image: url(../images/cart-green-icon.png);
    background-repeat: no-repeat;
    float: right;
    text-align: right;
    padding-right: 15px;
}
#cart.full
{
    width: 100%;
}
#cart.full *
{
    font-size: 18px;
}
#cart.full .total
{
    line-height: 36px;
    position: absolute;
    right: 180px;
}
#cart.fulltd.thumbnail
{
    width: 65px;
}
#cart.full .text
{
    text-indent: 10px;
    line-height: 1px;
}
.cart-detail-link
{
    color: #06F;
    font-weight: bold;
    text-align: left;
}
h1.cart-header:before
{
    content: url("../images/cart-header-icon.png");
}

```



Note: The code is styling the cart table and also the Cart webpage that will be built in the next exercise.

14. Save the file by pressing Ctrl+S.
15. Press Ctrl+F5 and look at the finished webpage.

Results: A home page that is written with HTML5 semantic elements.

Exercise 2: Creating the Game Details and Cart Pages

► Task 1: Create the Game Details webpage.

1. Right-click the **GameStore.Html** project in Solution Explorer.
2. Click the **Add** submenu, and then click **New Item**.
3. In the **Add New Item** dialog box, click **HTML Page**.
4. In the **Name** text box, type **GameDetails.html**
5. Click **Add** to add the new HTML file.
6. In the **GameDetails.html** file, delete the default title in the head element and add the following tags in the **head** element.

```
<meta charset="utf-8" />
<title>Game Details</title>
<link href="Styles/theme.css" rel="stylesheet" type="text/css" />
<link href="Styles/slideshow.css" rel="stylesheet" type="text/css" />
<link href="Styles/cart.css" rel="stylesheet" type="text/css" />
<link href="Styles/game.css" rel="stylesheet" type="text/css" />
<script src="Scripts/jquery-1.5.1.js" type="text/javascript"></script>
<script src="Scripts/GlobalFunctions.js" type="text/javascript"></script>
```

7. In the **body** element, create a **div** and set its class attribute to container.

```
<div class="container">
</div>
```

8. Create a **header** section inside the **div** element.

```
<header>
<p id="logo">Contoso, Ltd</p>
<nav>
<ul>
<li><a class="page-link active" href="Homepage.html">Games</a></li>
<li><a class="page-link" href="#">Checkout</a></li>
</ul>
<a class="go-to-cart-link money-text" href="#">Go to Cart</a>
</nav>
<h1>Game Details: Kinect Sports</h1>
</header>
```

9. Create a **section** after the **header** element. Set its class attribute to **game-slideshow**.

```
<section class="game-slideshow">
<div class="main-slide">

</div>
<div class="slide">

</div>
<div class="slide">

</div>
<div class="slide">

</div>
<div class="slide">

</div>
```

```
</div>
</section>
```

10. After the **game-slideshowsection**, create another **section**. Set the class attribute of the section to **game-box**.

```
<section class="game-box">
<article>
<figure class="game-thumbnail">

</figure>
<div class="game-title">
    Kinect Sports
</div>
<div class="game-subtitle">
    YOU ARE THE ATHLETE
</div>
<div class="game-description">
    Game description comes here
</div>
</article>
<div class="line"></div>
<div class="game-properties">
<label class="text">Genre:</label>
<div class="text">Sport</div>
<label class="text">Release Date:</label>
<time datetime="2011-08-11" class="text">11/08/2011</time>
<label class="text">Price:</label>
<div id="price" class="text">$30</div>
<label class="text">Game Rate:</label>
<div class="text">

</div>
</div>
<div class="button-bar">
<div class="button add-to-cart-button">
<a href="Cart.html">Add to Cart</a>
</div>
</div>
</section>
```

11. Create an **article** element after the **game-box section** and populate it with some sentences.

```
<article class="game-info">
    Game info comes here
</article>
```

12. After the **article**, create a **section** element. Set its class attribute to **chart-box**.

```
<section class="chart-box">

</section>
```

13. Create a **section** after the **chart-box section**. Set its class attribute to **game-reviews** and populate it with reviews.

```
<section class="game-reviews">
<header>
<h2>Reviews:</h2>
<a href="#" class="add-review-link">Add Review</a>
</header>
```

```

<div class="content">
<article class="game-review">
<q>Awesomgame</q>
<div class="nickname">
Justin Harrison
</div>
</article>
<article class="game-review">
<q>Awesomgame</q>
<div class="nickname">
Justin Harrison
</div>
</article>
<article class="game-review">
<q>Awesomgame</q>
<div class="nickname">
Justin Harrison
</div>
</article>
<article class="game-review">
<q>Awesome game<div class="nickname">
Justin Harrison
</div>
</article>
</div>
<footer>
<a href="#" class="add-review-link">Add Review</a>
</footer>
</section>

```

14. After the game-reviews section, create the footer.

```

<footer>
<p class="text">Copyright <a class="xbox-link" href="http://www.xbox.com">Xbox Site</a></p>
</footer>

```

15. Save the file by pressing Ctrl+S.
16. Right-click the **GameDetails.html** file.
17. Click the **View in Browser** menu item to see the completed webpage.

► Task 2: Create the Cart webpage.

1. Right-click the **GameStore.Html** project in Solution Explorer.
2. Click the **Add** submenu, and then click **New Item**.
3. In the **Add New Item** dialog box, click **HTML Page**.
4. In the **Name** text box, type **Cart.html**
5. Click **Add** to add the new HTML file.
6. In the **Cart.html** file, delete the default title in the **head** element and add the following tags in the **head** element.

```

<meta charset="utf-8" />
<title>Cart</title>
<link href="Styles/theme.css" rel="stylesheet" type="text/css" />
<link href="Styles/cart.css" rel="stylesheet" type="text/css" />
<script src="Scripts/jquery-1.5.1.js" type="text/javascript"></script>

```

```
<script src="Scripts/GlobalFunctions.js" type="text/javascript"></script>
```

7. In the **body** element, create a **div**. Set its class attribute to **container**.

```
<div class="container">
</div>
```

8. Create a **header** section inside the **div** element.

```
<header>
<p id="logo">Contoso, Ltd</p>
<nav>
<ul>
<li><a class="page-link active" href="Homepage.html">Games</a></li>
<li><a class="page-link" href="#">Checkout</a></li>
</ul>
<a class="go-to-cart-link money-text" href="#">Go to Cart</a>
</nav>
<h1>Game Details: Kinect Sports</h1>
</header>
```

9. Create a **section** after the **header** element. Set its **id** attribute to **cart**, and its **class** attribute to **box full**.

```
<section id="cart" class="box full">
</section>
```

10. Inside the **cartsection**, add another **header** element.

```
<header>
<h2>Games in Cart</h2>
</header>
```

11. Create a **section** under the previous header element from step 5. Set its class attribute to **content** and populate it with the cart table.

```
<section class="content">
<table>
<tr>
<td class="thumbnail">

</td>
<td class="text">Kinect Star Wars</td>
<td>
<a class="cart-detail-link" href="GameDetails.html">Details</a>
</td>
<td class="text">$30</td>
<td>

</td>
</tr>
<tr>
<td class="thumbnail">

</td>
<td>
<p class="text">Kinect Star Wars</p>
</td>
<td>
```



```

<a class="cart-detail-link" href="GameDetails.html">Details</a>
</td>
<td>
<p class="text">$30</p>
</td>
<td>

</td>
</tr>
<tr>
<td class="thumbnail">

</td>
<td>
<p class="text">Kinect Star Wars</p>
</td>
<td>
<a class="cart-detail-link" href="GameDetails.html">Details</a>
</td>
<td>
<p class="text">$30</p>
</td>
<td>

</td>
</tr>
</table>
</section>

```

12. Create a **footer** for the cart content under the **content section**.

```

<footer>
<div class="total">
    Total: <span id="totalPrice">$90</span>
</div>
<div class="cart-icon" />
</footer>

```

13. Create a **section** after the **cart section** that you created in step 4. Set its class attribute to **button-bar** and populate its content.

```

<section class="button-bar">
<div class="button">
<a href="#">Go to Checkout</a>
</div>
</section>

```

14. Create a **footer** after the **button-bar section**.

```

<footer>
<p class="text">Copyright <a class="xbox-link" href="http://www.xbox.com">Xbox Site</a></p>
</footer>

```

15. Save the file by pressing Ctrl+S.
16. Right-click the **Cart.html** file.
17. Click the **View in Browser** menu item to see the completed webpage.

► **Task 3: Add CSS styles to the Game Details webpage.**

1. In Solution Explorer, right-click the **Styles** folder under the **GameStore.Html** project.
2. Click the **Add** submenu, and then click **Existing Item**.
3. In the **Add Existing Item** dialog box, choose the **game.css** file that exists in **D:\Mod02\Labfiles\Starter\Css** folder.
4. Click the **Add** button to add the CSS file.

Results: Full game details and cart pages are written with HTML5 semantic elements.

Exercise 3: Adding Navigation and Menus to the Home Page

► Task 1: Add links from the Homepage to the other webpages.

1. In the **homepage.html** file, press Ctrl+F to open the **Find and Replace** dialog box.
2. Click the **Quick Replace** button.
3. In the **Find what:** text box, type the following code.

```
<a class="learn-more" href="#">Learn More ></a>
```

4. In the **Replace with:** text box, type the following code.

```
<a class="learn-more" href="GameDetails.html">Learn More ></a>
```

This will replace the # with a link to the new Game Details webpage.

5. Ensure that the Look in drop-down box is set to Current Document, and then click the **Replace All** button.
6. In the **a** tag in the **header** element, replace the # with **cart.html**.

```
<a class="go-to-cart-link money-text" href="Cart.html">Go to Cart</a>
```

7. Save the file by pressing Ctrl+S.
8. Press Ctrl+F5 to run the web application. You can now click the **Go to Cart** button in the homepage to navigate to the Cart page, and click the **Learn More** links to go to the Game Details page.

Results: Home, Game Details, and Cart pages that incorporate navigation and menus are created.

Module 3: Creating Form Input and Validation

Lab: Creating Form Input and Validation

Exercise 1: Creating the Checkout and Game Review Forms

► Task 1: Create the Checkout webpage.

1. Log on to the **SEA-DEV** virtual machine named as **Admin**, with the password, **Pa\$\$w0rd**.
2. Open Microsoft® Visual Studio® 2010.
3. Open the **GameStore.sln** solution from the **D:\Mod03\Labfiles\Starter** folder.
4. In Solution Explorer, right-click the **GameStore.Html** project.
5. Click the **Add** submenu, and then click **New Item**.
6. In the **Add New Item** dialog box, click **HTML Page**.
7. In the **Name** text box, type **Checkout.html**
8. Click **Add** to add the new HTML file.
9. In the **Checkout.html** file, remove the default **title** element and add the following content in the **head** element.

```
<meta charset="utf-8" />
<title>Checkout</title>
<link href="Styles/theme.css" rel="stylesheet" type="text/css" />
<link href="Styles/cart.css" rel="stylesheet" type="text/css" />
<link href="Styles/forms.css" rel="stylesheet" type="text/css" />
<script src="Scripts/jquery-1.5.1.js" type="text/javascript"></script>
<script src="Scripts/GlobalFunctions.js" type="text/javascript"></script>
```

10. In the **body** element, create a **div**. Set its class attribute to **container**.

```
<div class="container">
</div>
```

11. In the container **div**, add a **header** element and populate it with a logo, navigation area, and title.

```
<header>
<p id="logo">Contoso, Ltd</p>
<nav>
<ul>
<li><a class="page-link" href="Homepage.html">Games</a></li>
<li><a class="page-link active" href="Checkout.html">Checkout</a></li>
</ul>
<a class="go-to-cart-link money-text" href="Cart.html">Go to Cart</a>
</nav>
<h1>Checkout</h1>
</header>
```

12. Under the **header** element, create a form element.

```
<form id="mainForm" action="#" method="post">
</form>
```

13. Inside the **form** element, add two **fieldsets** for the buyer information and for the credit card information.

```
<fieldset class="box">
<legend>Buyer Information</legend>
<div>
</div>
</fieldset>
<fieldset class="box">
<legend>Credit Card Information</legend>
<div>
</div>
</fieldset>
```

14. Under the buyer information **fieldset**, enter the following code into the empty **div**.

```
<p>
<label class="field-name" for="FirstNameField">First Name</label>
<input id="FirstNameField" name="FirstName" type="text" class="field-value" size="10"
maxlength="50" />
</p>
<p>
<label class="field-name" for="LastNameField">Last Name</label>
<input id="LastNameField" name="LastName" type="text" class="field-value" size="10"
maxlength="50" />
</p>
<p>
<label class="field-name" for="EmailField">Email</label>
<input id="EmailField" name="Email" type="email" class="field-value" size="10" />
</p>
<p>
<label class="field-name" for="BirthDateField">Birth Date</label>
<input id="BirthDateField" name="BirthDate" type="date" class="field-value" />
</p>
<p>
<label class="field-name" for="AddressField">Shipping address</label>
<input id="AddressField" name="Address" type="text" class="field-value" />
</p>
```

15. Under the Credit Card Information fieldset, enter the following code into the empty **div**.

```
<p>
<label class="field-name" for="CCNumberField">Credit Card Number</label>
<input id="CCNumberField" name="CCNumber" type="text" class="field-value" />
</p>
<p>
<label class="field-name" for="CVVField">CVV </label>
<input id="CVVField" name="CVV" type="number" class="field-value" />
</p>
<p>
<label class="field-name" for="CardExpirationField">Card Expiration</label>
<input id="CardExpirationField" name="CardExpiration" type="month" class="field-value" />
</p>
```

16. After the last **fieldset**, add the following **aside** element.

```
<aside id="cart" class="box wide">
<header>
<h2>Games in Cart</h2>
</header>
<section class="content">
<table>
```

```

<tr>
<td class="thumbnail">

</td>
<td class="text">Kinect Star Wars</td>
<td class="text">$30</td>
<td>

</td>
</tr>
<tr>
<td class="thumbnail">

</td>
<td class="text">Kinect Star Wars</td>
<td>
<p class="text">$30</p>
</td>
<td>

</td>
</tr>
<tr>
<td class="thumbnail">

</td>
<td class="text">Kinect Star Wars</td>
<td>
<p class="text">$30</p>
</td>
<td>

</td>
</tr>
</table>
</section>
<footer>
<div class="total">
Total: <span id="totalPrice">$90</span>
</div>
<a href="Cart.html">
<div class="cart-icon"></div>
</a>
</footer>
</aside>

```

17. After the **aside** element, add a **section** containing a purchase button.

```

<section class="button-bar">
<div class="button">
<a href="#">Purchase</a>
</div>
</section>

```

18. After the purchase button **section**, place a **footer** element that exists in the footer.txt file that is located in the **D:\Mod03\Labfiles\Starter\Assets** folder.

19. Save the file by pressing Ctrl+S.

20. Right-click the **Checkout.html** file.

21. Click the **View in Browser** menu item to see the completed webpage.

► **Task 2: Create the Game Review webpage.**

1. In Solution Explorer, right-click the **GameStore.Html** project.
2. Click the **Add** submenu, and then click **New Item**.
3. In the **Add New Item** dialog box, click **HTML Page**.
4. In the **Name** text box, name the page **GameReview.html**.
5. Click **Add** to add the new HTML file.
6. In the **GameReview.html** file, remove the default **title** element and add the following content in the **head** element.

```
<meta charset="utf-8" />
<title>Game Review</title>
<link href="Styles/theme.css" rel="stylesheet" type="text/css" />
<link href="Styles/cart.css" rel="stylesheet" type="text/css" />
<link href="Styles/game.css" rel="stylesheet" type="text/css" />
<link href="Styles/forms.css" rel="stylesheet" type="text/css" />
<script src="Scripts/jquery-1.5.1.js" type="text/javascript"></script>
<script src="Scripts/GlobalFunctions.js" type="text/javascript"></script>
```

7. In the **body** element, create a **div**. Set its class attribute to **container**.

```
<div class="container">
</div>
```

8. In the **containerdiv**, add a **header** element and populate it with a logo, navigation area, and title.

```
<header>
<p id="Logo">Contoso, Ltd</p>
<nav>
<ul>
<li><a class="page-link active" href="Homepage.html">Games</a></li>
<li><a class="page-link" href="Checkout.html">Checkout</a></li>
</ul>
<a class="go-to-cart-link money-text" href="Cart.html">Go to Cart</a>
</nav>
<h1>Submit Review: Kinect Sports</h1>
</header>
```

9. Under the **header** element, create a **form** element.

```
<form id="mainForm" action="#" method="post">
</form>
```

10. Inside the **form** element, create a **fieldset** for the review.

```
<fieldset class="box">
<legend>Your Review</legend>
<div>
</div>
</fieldset>
```

11. Populate the empty **div** with the following code.

```
<p>
```

```

<label class="field-name" for="FirstNameField">First Name</label>
<input id="FirstNameField" name="FirstName" type="text" class="field-value" size="10"
maxlength="50" />
</p>
<p>
<label class="field-name" for="LastNameField">Last Name</label>
<input id="LastNameField" name="LastName" type="text" class="field-value" size="10"
maxlength="50" />
</p>
<p>
<label class="field-name" for="ReviewField">Review</label>
<textarea id="ReviewField" name="Review" class="field-value" maxlength="4000"></textarea>
</p>

```

12. After the **form** element, create the following **section**.

```

<section class="game-box">
<article>
<figure class="game-thumbnail">

</figure>
<div class="game-title">
    Kinect Sports
</div>
<div class="game-subtitle">
    YOU ARE THE ATHLETE
</div>
<div class="game-description">
    Season Two sprints out the box with six new sports and exciting new features that
    take full advantage of the revolutionary Kinect technology. The second brings families
    and the world together like never before with the addition of Challenge Play to
    Xbox LIVE and harnesses the power of friendly competition to provide a more
    immersive
    experience for everyone from casual players to top-notch athletes
</div>
</article>
<div class="line"></div>
<div class="game-properties">
<label class="text">Genre:</label>
<div class="text">Sport</div>
<label class="text">Release Date:</label>
<time datetime="2011-08-11" class="text">11/08/2011</time>
<label class="text">Price:</label>
<div id="price" class="text">$30</div>
<label class="text">Game Rate:</label>
<div class="text">

</div>
</div>
<div class="button-bar">
<div class="button add-to-cart-button">
<a href="Cart.html">Add to Cart</a>
</div>
</div>
</section>

```

13. After the preceding **section**, add a buttons **section**.

```

<section class="button-bar">
<div class="button">
<a href="#">Submit</a>
</div>
<div class="button">

```



```
<a href="GameDetails.html">Cancel</a>
</div>
</section>
```

14. After the purchase button section, place a footer element that exists in the footer.txt file that is located in the D:\Mod03\Labfiles\Starter\Assets folder.
15. Save the file by pressing Ctrl+S.
16. Right-click the **GameReview.html** file.
17. Click the **View in Browser** menu item to see the completed webpage.

► **Task 3: Add CSS styles to the Checkout and Game Review webpages.**

1. In Solution Explorer, right-click the **Styles** folder under the **GameStore.Html** project.
2. Click the **Add** submenu, and then click **Existing Item**.
3. In the **Add Existing Item** dialog box, choose the **forms.css** that exists in **D:\Mod03\Labfiles\Starter\Css** folder.
4. Click the **Add** button to add the CSS file.

► **Task 4: Add navigation to the new webpages.**

1. Open the **Homepage.html** file.
2. Inside the unordered list (the **ul** element), find the list item (the **li** element), which is located in the header and whose text content is "Checkout", and change the # in the **href** attribute to **Checkout.html**.
3. Save the file by pressing Ctrl+S.
4. Open the **GameDetails.html** file.
5. Inside the unordered list (the **ul** element), find the list item (the **li** element), which is located in the header and whose text content is "Checkout", and change the # in the **href** attribute to **Checkout.html**.
6. Save the file by pressing Ctrl+S.
7. Open the **Cart.html** file.
8. Inside the unordered list (the **ul** element), find the list item (the **li** element), which is located in the header and whose text content is "Checkout", and change the # in the **href** attribute to **Checkout.html**.
9. Save the file by pressing Ctrl+S.
10. Open the **GameDetails.html** file.
11. In **GameDetails.html**, place the cursor at the beginning of the file, and then press Ctrl+F to open the **Find and Replace** dialog box.
12. In the **Find and Replace** dialog box, type **Add Review**.
13. Find the first **anchor** that will browse to the **Game Review** page by pressing **Enter**.
14. In the **anchor's href** attribute, replace the # with **GameReview.html**.
15. Press Ctrl+F again.

16. In the **Find and Replace** dialog box, type **Add Review**.
 17. Find the second **anchor** that will browse to the **Game Review** page by pressing **Enter**.
 18. In the **href** attribute of that **anchor**, replace the # with **GameReview.html**.
 19. Save the file by pressing Ctrl+S.
 20. Press Ctrl+F5 and run the application.
- Now, you can navigate to all the webpages that you created.

Results: Checkout and Game Review Forms are created by utilizing HTML5 input types.

Exercise 2: Add validation to the Checkout and Game Review forms.

► Task 1: Add validation and HTML5 attributes to the Checkout webpage.

1. In the **Checkout.html** file, in the buyer information **fieldset**, add the **required**, **validation**, and **placeholder** attributes to the fields, and an **autofocus** for the first field. Your code should look like the following code after you finish.

```
<p>
<label class="field-name" for="FirstNameField">First Name</label>
<input id="FirstNameField" name="FirstName" type="text" autofocus="autofocus"
required="required" placeholder="Enter your first name" class="field-value" size="10"
maxlength="50" />
</p>
<p>
<label class="field-name" for="LastNameField">Last Name</label>
<input id="LastNameField" name="LastName" type="text" required="required" placeholder="Enter
your last name" class="field-value" size="10" maxlength="50" />
</p>
<p>
<label class="field-name" for="EmailField">Email</label>
<input id="EmailField" name="Email" type="email" required="required" placeholder="Enter your
email address" class="field-value" size="10" />
</p>
<p>
<label class="field-name" for="BirthDateField">Birth Date</label>
<input id="BirthDateField" name="BirthDate" type="date" required="required" class="field-value"
/>
</p>
<p>
<label class="field-name" for="AddressField">Shipping address</label>
<input id="AddressField" name="Address" type="text" required="required" placeholder="Enter the
shipping address" class="field-value" />
</p>
```

2. In the **Checkout.html** file, in the credit card information **fieldset**, add the **required**, **validation**, and **placeholder** attributes to the fields, and an **autofocus** for the first field. Your code should look like the following code after you finish.

```
<p>
<label class="field-name" for="CCNumberField">Credit Card Number</label>
<input id="CCNumberField" name="CCNumber" type="text" required="required" placeholder="Enter
your credit card number" class="field-value" />
</p>
<p>
<label class="field-name" for="CVVField">CVV </label>
<input id="CVVField" name="CVV" type="number" required="required" class="field-value" />
</p>
<p>
<label class="field-name" for="CardExpirationField">Card Expiration</label>
<input id="CardExpirationField" name="CardExpiration" type="month" class="field-value" />
</p>
```

3. Save the file by pressing Ctrl+S.

Results: The Checkout and Game Review forms with validation attributes are created.

Exercise 3: Adding Modernizr Support

► Task 1: Add the Modernizr JavaScript library to the project.

1. In the **GameStore.Html** project, right-click the **Scripts** folder.
2. Click the **Add** submenu, and then click **Existing Item**.
3. In the **Add Existing Item** dialog box, insert **D:\Mod03\Labfiles\Assets** in the File name text box.
4. Press **Enter** to navigate to the folder.
5. Click the **modernizr-1.7.js** file.
6. Click **Add** to add the **Modernizr** file to the project.
7. In the **Homepage.html** file, add the following script at the end of the **head** element.

```
<script src="Scripts/modernizr-1.7.js" type="text/javascript"></script>
```

8. Save the file by pressing Ctrl+S.
9. In the **Checkout.html** file, add the following script at the end of the **head** element.

```
<script src="Scripts/modernizr-1.7.js" type="text/javascript"></script>
```

10. Save the file by pressing Ctrl+S.
11. In the **GameReview.html** file, add the following script at the end of the **head** element.

```
<script src="Scripts/modernizr-1.7.js" type="text/javascript"></script>
```

12. Save the file by pressing Ctrl+S.

► Task 2: Create the number input type and placeholder attribute polyfills.

1. In the **Scripts** folder, open the **GlobalFunctions.js** file.
2. Add the following statement after the comment that references the jQueryvsdoc file to enable **IntelliSense**.

```
/// <reference path="modernizr-1.7.js" />
```

3. Create a new JavaScript function under the **contentLoaded** function and call it **addPolyfills**.

```
function addPolyfills() {
}
```

4. Add the following code inside the **addPolyfills** function.

```
if (!Modernizr.inputtypes.number) {
// grab all the number input types and wire the key press event
$('input[type=number]').keypress(function (e) {
// get the key code and check whether it is a number
var charCode = e.keyCode;
if (charCode > 31 && (charCode < 48 || charCode > 57)) {
return false;
}
return true;
});
}
```

```
if (!Modernizr.input.placeholder) {  
    // grab all the input types with placeholder attribute  
    $('input[placeholder]').each(function () {  
        // for each input type insert the content of the placeholder attribute to the value of the  
        input type      $(this).val(this.getAttribute('placeholder'));  
    });  
}
```

5. Add a call to the **addPolyfills** function at the beginning of the **contentLoaded** function. The function should look like the following code.

```
function contentLoaded() {  
    addPolyfills();  
    startSlideShow();  
    addCollapsiblePanels();  
    collapseAllPanels();  
    addSubmitBehavior();  
}
```

6. Save the file by pressing Ctrl+S.
7. In Solution Explorer, select the **Homepage.html** file.
8. Press Ctrl+F5 and run the application.
9. Browse to the Checkout form by clicking the **Checkout** button.
10. Open **IE Developer Tools** by pressing F12 in the browser.
11. Select **Browser Mode** on the Developer Tools menu.
12. Choose **Internet Explorer 9** to see the polyfills in action.

Results: The form pages to utilize the Modernizr library are created.

Module 4: Laying Out and Styling Webpages

Lab: Laying Out and Styling Webpages

Exercise 1: Add a CSS3 style to the store's webpages.

► Task 1: Create the new CSS3 style file.

1. Log on to the virtual machine named **SEA-DEV** as **Admin** with the password **Pa\$\$w0rd**
2. Open Microsoft® Visual Studio® 2010.
3. Open **D:\Mod04\Labfiles\Starter** folder, and then open **GameStore.sln**.
4. In **Solution Explorer**, right-click the **Styles** folder under the **GameStore.Html** project.
5. Click the **Add** submenu, and then click **New Item**.
6. In the **Add New Item** dialog box, click **Style Sheet**.
7. In the **Name** text box, type **css3.css**.
8. Click **Add**. The new CSS file is added.
9. Double-click **Homepage.html**.
10. In the head section, under the css file links, add the following link.

```
<link href="Styles/css3.css" rel="stylesheet" type="text/css" />
```

11. Press Ctrl+S. The file is saved.
12. Repeat steps 1-3 on all the other webpages in the solution (Cart.html, Checkout.html, GameDetails.html, and GameReview.html).

► Task 2: Add the @font-face rule support.

1. In the **css3.css** file, add the following @font-face definition.

```
@font-face
{
font-family: myFont;
src: url(../Styles/Action_Man-webfont.woff);
}
```

2. Add the font to the **game-text** class by adding the following code after the **@font-face** definition.

```
.game-text
{
font-family: myFont;
}
```

3. In **Solution Explorer**, right-click the **Homepage.html** file (located in **Styles** folder) and choose **Set As Start Page**.
4. Press Ctrl+S. The file is saved.
5. Press Ctrl+F5. The application runs.
6. Look at the games titles in the genres section.

► **Task 3: Add a box-shadow to the logo and the footer.**

1. In the **css3.css** file, add the following box-shadow property.

```
#logo, .container > footer
{
    box-shadow: 0px 0px 10px #888;
}
```

2. Press Ctrl+S. The file is saved.
3. Press Ctrl+F5. The application runs.
4. Look at the header and footer sections.

► **Task 4: Add a Media Query property.**

1. In the **css3.css** file, add the following media query property.

```
/* media query to remove the box-shadow
   if the width of the browser greater
   than 400 pixel the shadow will be shown */
@media screen and (max-width:400px)
{
    #logo, .container > footer
    {
        box-shadow: 0px 0px 0px #888;
    }
}
```

2. Press Ctrl+S. The file is saved.
3. Press Ctrl+F5. The application runs.
4. Resize the browser window to 400 pixels.

► **Task 5: Add border-radius support.**

1. In the **css3.css** file, add the following CSS code.

```
nav, .game-box
{
    border-radius: 10px;
}
#logo, legend, .box > header
{
    border-top-left-radius: 10px;
    border-top-right-radius: 10px;
}
.container > footer, #cart > footer, fieldset > div
{
    border-radius: 0px 0px 10px 10px;
}
```

2. Press Ctrl+S. The file is saved.
3. Press Ctrl+F5. The application runs.
4. Look at the header, footer, and cart rounded corners.

► **Task 6: Add the nth-child pseudo element.**

1. In the **css3.css** file, add the following CSS code.

```
#cart table tr:nth-child(odd)
{
    background-color: #e5e5e5;
}
```

2. Press Ctrl+S. The file is saved.
3. Press Ctrl+F5. The application runs.
4. Look at the color of the odd rows in the cart.

► Task 7: Add multi-columns Support.

1. In the **css3.css** file, add the following CSS code.

```
.game-info
{
    column-count: 3;
    column-rule: solid 1px lightgray;
    column-gap: 30px;
}
```

2. Press Ctrl+S. The file is saved.
3. Press Ctrl+F5. The application runs.
4. Look at the game info section in the **GameDetails.html** webpage. Use the link *Learn More* > on the homepage to navigate to that page.

► Task 8: Add background-size support.

1. In the **css3.css** file, add the following CSS code.

```
.button
{
    background-size: 100% 100%;
}
```

2. Press Ctrl+S. The file is saved.
3. Press Ctrl+F5. The application runs.
4. Look at the buttons on the homepage.

► Task 9: Add CSS3 animation.

1. In the **css3.css** file, add the following @keyframes move rule after the CSS code you wrote in the previous task.

```
@keyframes move
{
    from { color: White; padding-left: 10px; }
    50% { padding-left: 0px; }
    to { color: #2a801f; padding-left: 5px; }
}
```

2. Apply the animation on the logo element by adding the following code to the CSS file after the @keyframes declaration.

```
#logo
{
```



```
animation: move 2s linear 0s infinite;  
}
```

3. Press Ctrl+S. The file is saved.
4. Press Ctrl+F5. The application runs.
5. Look at logo on the homepage.

Results: CSS3 elements are integrated onto the store's webpages.

Module 5: Getting Started with Graphics and Multimedia Elements

Lab: Getting Started with Graphics and Multimedia Elements

Exercise 1: Adding Video to the Details Page

► **Task 1: Replace the large image with a new video element.**

1. Log on to the virtual machine named **SEA-DEV** as **Admin** with the password **Pa\$\$w0rd**.
2. Open Microsoft® Visual Studio® 2010.
3. Open the **GameStore.sln** solution from the **D:\Mod05\LabFiles\Starter** folder.
4. In case a message box appears prompting you to create a virtual directory for the project click **Yes**.
5. In Solution Explorer, expand the **GameStore.Mvc** project node if not already expanded, and navigate to the **Views\Game** folder.
6. In the **Views\Game** folder, open the **Details.cshtml** file.
7. Find the **div** element with the "main-slide" **class** attribute. The div looks like the following code.

```
<div class="main-slide">
  
</div>
```

8. Remove the **img** element from inside the **div** element. The **div** element should now be empty.

```
<div class="main-slide"></div>
```

9. Add a new empty **video** element inside the **div** with the "main-slide" class.

```
<video></video>
```

10. Set the **video** element's **src** attribute.

```
<video src="/ContosoGames/Content/Videos/Games/1/trailer1.m4v"></video>
```

11. Save the file by pressing Ctrl+S.
12. In Solution Explorer, expand the **Content\Styles** folder and open the **game.css** file.
13. Find the selector, **.main-slide**. The style rule should look like the following code.

```
.main-slide
{
    width: 560px;
    height: 240px;
    margin-bottom: 10px;
}
```

14. Remove the **height** property and its value. This will allow the video to resize vertically on its own.

```
.main-slide
```

```
{
    width: 560px;
    margin-bottom: 10px;
}
```

15. Add another CSS style rule declaration for the **video** element right after the **.main-slide** rule to ensure the element has the correct width.

```
.main-slide video
{
    width: 560px;
}
```

16. Save and close the **game.css** file.
17. Run the application by pressing F5 in Visual Studio.
18. In Internet Explorer, navigate to the **/Game/Details** page by clicking the **Learn More** link under the **Add to Cart** button for the **Kameo: Elements of Power** game.
19. Observe the new slideshow with the **video** element.

► **Task 2: Add support for multiple video sources.**

1. Remove the **src** attribute from the **video** element.

```
<video></video>
```

2. Add three source elements, one for each codec (*MP4/H.264*, *WebM/VP8* and *Ogg/Theora*) inside the **video** element.

```
<video>
  <source src="/ContosoGames/Content/Videos/Games/1/trailer1.m4v" type="video/mp4" />
  <source src="/ContosoGames/Content/Videos/Games/1/trailer1.webm" type="video/webm" />
  <source src="/ContosoGames/Content/Videos/Games/1/trailer1.ogv" type="video/ogg" />
</video>
```

► **Task 3: Set the video element's behavior.**

1. Add the Boolean **autoplay** attribute to the **video** element.

```
<video controls="controls" autoplay="autoplay">...</video>
```

2. Add the Boolean **muted** attribute to the **video** element.

```
<video controls="controls" autoplay="autoplay" muted="muted">...</video>
```

3. Add the Boolean **loop** attribute to the **video** element.

```
<video controls="controls" autoplay="autoplay" muted="muted" loop="loop">...</video>
```

4. Run the application by pressing F5 in Visual Studio.
5. In Internet Explorer, navigate to the **/Game/Details** page by clicking the **Learn More** link under the **Add to Cart** button for the **Kameo: Elements of Power** game.
6. Observe the **video** element with built-in controls, and automatic playback and looping behavior.

► **Task 4: Add fallback behavior.**

1. Add the **poster** attribute to the **video** element and set the **poster** attribute value to the path of the image that was removed earlier.

```
<video controls="controls" autoplay="autoplay" muted="muted" loop="loop"
poster="/ContosoGames/Content/Images/game-details.jpg">...</video>
```

2. Add the **img** element that was previously removed as the **video** element's last child, after the three **source** elements, to serve as fallback on browsers that do not support the **video** element.

```

```

3. The **video** element should now look like the following code.

```
<video controls="controls" autoplay="autoplay" muted="muted" loop="loop"
poster="/ContosoGames/Content/Images/game-details.jpg">
  <source src="/ContosoGames/Content/Videos/Games/1/trailer1.m4v" type="video/mp4" />
  <source src="/ContosoGames/Content/Videos/Games/1/trailer1.webm" type="video/webm" />
  <source src="/ContosoGames/Content/Videos/Games/1/trailer1.ogv" type="video/ogg" />
  
</video>
```

Results: A Game Details webpage with a **video** element is displayed.

Exercise 2: Creating a Slideshow by Using the Canvas

► Task 1: Create a canvas element for the slideshow.

1. In the **Details.cshtml** file, immediately before the "main-slide" div element, add an empty **canvas** element:

```
<canvas></canvas>
```

2. Set the **canvas** element's **id** to "slideshow".

```
<canvas id="slideshow"></canvas>
```

3. Set the **canvas** element's size to 560 pixels wide and 400 pixels high.

```
<canvas id="slideshow" width="560" height="400"></canvas>
```

4. In Solution Explorer, expand the **Content\Styles** folder and open the **game.css** file.
5. Find the **.main-slide** selector. It should look like the following code.

```
.main-slide
{
    width: 560px;
    margin-bottom: 10px;
}
```

6. Add another style rule declaration immediately before it to set the **canvas** element's grid size.

```
#slideshow
{
    width: 560px;
}
```

7. Save and close the **game.css** file

► Task 2: Add fallback behavior.

1. In the **Details.cshtml** file, find the section with the class **game-slideshow**. Its opening looks like the following snippet.

```
<section class="game-slideshow">
```

2. Its first child is the **canvas** element we created in the preceding task. Move the **canvas** element's closing tag to the end of the **game-slideshow** section, so that the canvas is the only child of the section and contains all the slides. The section element should look like the following code.

```
<section class="game-slideshow">
    <canvas id="slideshow" width="560" height="400">
        <div class="main-slide">
            <video controls="controls" autoplay="autoplay" muted="muted" loop="loop"
poster="/ContosoGames/Content/Images/game-details.jpg">
                <source src="/ContosoGames/Content/Videos/Games/1/trailer1.m4v"
type="video/mp4" />
                <source src="/ContosoGames/Content/Videos/Games/1/trailer1.webm"
type="video/webm" />
            </div>
        </canvas>
    </section>
```

```

        <source src="/ContosoGames/Content/Videos/Games/1/trailer1.ogv"
type="video/ogg" />
        
    </video>
</div>
<div class="slide">
    
</div>
<div class="slide">
    
</div>
<div class="slide">
    
</div>
<div class="slide">
    
</div>
</canvas>
</section>

```



Note: Ensure that the **canvas** element's closing tag appears immediately before the **section** element's closing tag.

3. The previous content is now the **canvas** element's fallback content. It will also be used as the content that will be rendered on the canvas by using JavaScript.

► Task 3: Initialize the slideshow.

1. In Solution Explorer, navigate to the **Scripts** folder and open the **GameSlideshowScripts.js** file.
2. In the **initializeCanvas** function, use the **document.getElementById** method to find the **canvas** element.

```

function initializeSlideshow() {
    // find the canvas element
    var canvas = document.getElementById("slideshow");
}

```

3. After finding the **canvas** element, immediately before the end of the **initializeCanvas** function, call the **canvas** object's **getContext** method to get a 2D drawing context, and store the drawing context on the **slideshow** object's **context** property for later use.

```

// get a 2D drawing context and store it for later use
slideshow.context = canvas.getContext("2d");

```

4. In the **createVideoElement** function, create a **video** element, assign the **video** element's **src** attribute, set the video to automatically start playback, and mute the video.

```

function createVideoElement(urlInfo) {
    // create and initialize a video element
    var video = document.createElement("video");
    video.src = urlInfo.Url;
    video.autoplay = true;
    video.muted = true;
    return video;
}

```

5. In the **createImageElement** function, create an **img** element for a specific URL.

```
function createImageElement(urlInfo) {
    // create and initialize an image element
    var image = new Image();
    image.src = urlInfo.Url;
    return image;
}
```

► **Task 4: Draw outlines for the slides on the canvas.**

1. In the **GameSlideShowScripts.js** file, in the empty **drawMainSlide** function, call the drawing context's **strokeRect** method to draw the main slide's outline:

```
function drawMainSlide() {
    // draw the outline
    slideshow.context.strokeRect(0, 0, slideshow.mainSlideWidth, slideshow.mainSlideHeight);
}
```

2. In the empty **drawThumbSlides** function, retrieve an array of the thumbnail slides, get each slide's location, and then draw an outline for each slide by calling the drawing context's **strokeRect** method with the corresponding location info.

```
function drawThumbSlides() {
    // retrieve all the thumbnail slide elements
    var thumbSlides = $(slideshow.context.canvas).find(".slide");
    // for each retrieved thumbnail slide element, draw the slide's content
    $.each(thumbSlides, function (index, thumbSlide) {
        // determine where the slide should be drawn
        var location = getThumbSlideLocation(index);
        // draw the slide's outline
        slideshow.context.strokeRect(location.left, location.top, location.width,
        location.height);
    });
}
```

► **Task 5: Draw the thumbnail slides on the canvas.**

1. In the **drawImage** function, draw **img** elements on the canvas by calling the drawing context's **drawImage** method.

```
function drawImage(element, left, top, width, height) {
    var image = element;
    slideshow.context.drawImage(image, left, top, width, height);
}
```

2. In the **drawImage** function, check if the image has finished loading. If it has, call the drawing context's **drawImage** method. If it has not, wait 100 milliseconds and then call the **drawImage** function to retry. Replace the code in the **drawImage** function with the following code.

```
function drawImage(element, left, top, width, height) {
    var image = element;
    if (!image.complete) {
        // if the image has not yet finished loading, try again in 100 milliseconds
    }
}
```

```

    var redraw = function () {
        drawImage(image, left, top, width, height);
    };
    window.setTimeout(redraw, 100);
}
else {
    // draw the image onto the canvas
    slideshow.context.drawImage(image, left, top, width, height);
}
}

```

3. In the **drawThumbSlides** function, immediately after the call to the **slideshow.context.strokeRect** method, add the following code to call the **drawImage** function only for slides that contain **img** elements.

```

// draw the slide itself
var element = thumbSlide.firstChild;
if (element) {
    if (element instanceof HTMLImageElement) {
        // if the slide contains a static image, call the drawImage function
        drawImage(element, location.left, location.top, location.width, location.height);
    }
}

```

4. In the **drawVideo** function, call the drawing context's **drawImage** method to draw **video** elements on the canvas.

```

function drawVideo(element, left, top, width, height) {
    var context = slideshow.context;
    var video = element;
    context.drawImage(video, left, top, width, height);
}

```

5. In the **drawThumbSlides** function, after the **if** block that checks for **HTMLImageElements**, add a call to the **drawVideo** function only for slides that contain **video** elements.

```

if (element instanceof HTMLImageElement) {
    ...
}
else if (element instanceof HTMLVideoElement) {
    // if the slide contains video, start playing the video and call the drawVideo function
    element.play();
    drawVideo(element, location.left, location.top, location.width, location.height);
}

```

6. In the **drawVideo** function, after calling the **context.drawImage** method, create an anonymous function that calls the **drawVideo** function, and use the **window.setTimeout** method to call the anonymous function after 20 seconds, effectively creating the time loop.

```

function drawVideo(element, left, top, width, height) {
    var context = slideshow.context;
    var video = element;
    context.drawImage(video, left, top, width, height);
    // repaint the canvas with the updated video every 20 milliseconds
    var redraw = function () {
        // use an anonymous function to pass the element and the location
        drawVideo(video, left, top, width, height);
    };
}

```



```

window.setTimeout(redraw, 20);
}

```

7. In the **drawVideo** function, move the call to the **context.drawImage** method into an **if** block that checks if the video is playing, to ensure a paused or ended video is not redrawn unnecessarily.

```

// if the video is still playing, draw the current frame onto the canvas
if (!video.paused && !video.ended) {
    context.drawImage(video, left, top, width, height);
}

```

► **Task 6: Switch the main slide at regular intervals.**

1. In the **rotateSlide** function, switch to the next slide, or to the first slide if the current slide is the last slide.

```

function rotateSlide() {
    // switch the selected slide to the next slide
    // the modulus (%) operator is used to cycle back to the beginning
    slideshow.currentSlideIndex = (slideshow.currentSlideIndex + 1) % slideshow.slideCount;
}

```

2. In the **startSlideRotationTimeLoop** function, use the **window.setInterval** method to trigger the **rotateSlide** function at regular intervals.

```

function startSlideRotationTimeLoop() {
    // start a timer that calls the rotateSlide function periodically every
    "slideshow.interval" milliseconds
    window.setInterval(rotateSlide, slideshow.interval);
}

```

3. In the **drawMainSlide** function, after the call to the **slideshow.context.strokeRect** method, draw the current thumbnail slide as the main slide by copying the part of the canvas that contains the thumbnail to the main slide, by using the drawing context's **drawImage** method.

```

// copy the snapshot from the active thumbnail
var location = getThumbSlideLocation(slideshow.currentSlideIndex);
slideshow.context.drawImage(slideshow.context.canvas, location.left, location.top,
location.width, location.height, 0, 0, slideshow.mainSlideWidth, slideshow.mainSlideHeight);

```

4. At the end of the **drawMainSlide** function, add the following code to trigger an automatic refresh to ensure that the video thumbnail slides are also redrawn on the main slide.

```

// trigger the next iteration
window.setTimeout(drawMainSlide, 20);

```

► **Task 7: Toggle video playback when the user clicks a slide.**

1. In the **hitTest** function, check if the user clicked inside a specific area.

```

function hitTest(x, y, bounds) {
    // returns true only if the coordinates represented by x and y
    // are inside the rectangular area represented by bounds
    return bounds.left < x && bounds.left + bounds.width > x && bounds.top < y && bounds.top +
    bounds.height > y;
}

```

2. In the **handleClicked** function, which is called when the user clicks the canvas, inside the **if** block that determines if the clicked slide is a **video** element, toggle the state of the video between *playing* and *paused*.

```
if (element && element instanceof HTMLVideoElement) {
    if(element.paused || element.ended) {
        element.play();
    }
    else {
        element.pause();
    }
}
```

3. In the **addCanvasClickEventHandler** function, use jQuery to bind the **canvas** element's **click** event to the **handleClicked** function, passing it the click coordinates.

```
function addCanvasClickEventHandler() {
    // pass the mouse coordinates to the handleClick function when the user clicks the canvas
    $(slideshow.context.canvas).click(function (e) {
        handleClick(e.offsetX, e.offsetY);
    });
}
```

4. In the **drawVideo** function, after the **if** block that determines whether to draw the current video frame, and before the declaration of the **redraw** variable, use the Canvas API to draw a path shaped like a play symbol (a right-facing triangle) on the lower-right of the video when the video is not playing.

```
if (video.paused || video.ended) {
    // draw a play symbol so the user knows to click to play the video
    context.beginPath();
    context.moveTo(left + width * 0.9, top + height * 0.8);
    context.lineTo(left + width * 1.0, top + height * 0.9);
    context.lineTo(left + width * 0.9, top + height * 1.0);
    context.fillStyle = "rgba(0, 120, 0, 0.5)";
    context.fill();
}
```

5. Immediately after the previous block, add an **else** block that draws a pause symbol (two vertical bars) when the video is playing.

```
else {
    // draw a pause symbol so the user knows to click to pause the video
    context.fillStyle = "rgba(30, 30, 30, 0.5)";
    context.fillRect(left + width * 0.9, top + height * 0.8, width * 0.04, height * 0.2);
    context.fillRect(left + width * 0.96, top + height * 0.8, width * 0.04, height * 0.2);
}
```

6. Run the application by pressing F5 in Visual Studio.
7. In Internet Explorer, navigate to the **/Game/Details** page by clicking the **Learn More** link under the **Add to Cart** button for the **Kameo: Elements of Power** game.

8. Observe the new slideshow with the **canvas** element, playing video and animation.

Results: A Game Details webpage is displayed with a **canvas** element that plays a video.

Module 6: Creating Advanced Graphics

Lab: Creating Advanced Graphics

Exercise 1: Adding an SVG Chart with Animation

► Task 1: Add an SVG chart to the Game Details webpage.

1. Log on to the virtual machine named **SEA-DEV** as **Admin** with the password **Pa\$\$w0rd**.
2. Open Microsoft Visual Studio 2010®.
3. Open the **GameStore.sln** solution from the **D:\Mod06\Labfiles\Starter** folder.
4. In Solution Explorer, expand the **GameStore.Mvc** project node if it is not already expanded, and navigate to the **Views\Game** folder.
5. In the **Views\Game** folder, open the **Details.cshtml** file.
6. Inside the **svg** element, add a **text** element with the chart's title as the first child of the **svg** element.

```
<text x="100" y="15" class="title-text">Game Sales per Quarter</text>
```

7. Add a **line** element for the x-axis right after the **text** element.

```
<line x1="380" y1="220" x2="10" y2="220" class="x-axis-line"></line>
```

8. Immediately after the x-axis's **line** element, add four **text** elements, one for each quarter.

```
<text x="35" y="230" class="axis-text">Jan-Mar</text>
<text x="125" y="230" class="axis-text">Apr-Jun</text>
<text x="215" y="230" class="axis-text">Jul-Sep</text>
<text x="305" y="230" class="axis-text">Oct-Dec</text>
```

9. After the last text element, add a perpendicular **line** element for the y-axis.

```
<line x1="17" y1="220" x2="17" y2="0" class="y-axis-line"></line>
```

10. Add two marker **text** elements after the y-axis's **line** element.

```
<text x="3" y="150" class="axis-text">30</text>
<text x="3" y="70" class="axis-text">60</text>
```

► Task 2: Add data and animation to the bar chart.

1. After the last element inside the **svg** element, add four **rect** elements representing the sales per quarter data.

```
<rect x="20" y="20" width="70" height="200" rx="2" ry="2" class="bar"></rect>
<rect x="110" y="180" width="70" height="40" rx="2" ry="2" class="bar"></rect>
<rect x="200" y="120" width="70" height="100" rx="2" ry="2" class="bar"></rect>
<rect x="290" y="140" width="70" height="80" rx="2" ry="2" class="bar"></rect>
```

2. Inside the first rect element, add two child **animate** elements, one to animate the **y** attribute and the second to animate the **height** attribute.

```
<animate attributeName="y" fill="freeze" dur="3" begin="2" />
```

```
<animate attributeName="height" fill="freeze" dur="3" begin="2" />
```

3. Add the **to** attribute to the first **animate** element and move the **rect** element **y** attribute to the **animate** element's **to** attribute. Also set **y** and **height** to appear above the axis at the start.

```
<rect x="20" y="215" width="70" height="200" rx="2" ry="2" class="bar">
  <animate attributeName="y" fill="freeze" dur="3" begin="2" to="20" />
  <animate attributeName="height" fill="freeze" dur="3" begin="2" />
</rect>
```

4. Do the same for the **rect** element's **height** attribute and the height **animate** element's **to** attribute.

```
<rect x="20" y="215" width="70" height="5" rx="2" ry="2" class="bar">
  <animate attributeName="y" fill="freeze" dur="3" begin="2" to="20" />
  <animate attributeName="height" fill="freeze" dur="3" begin="2" to="200" />
</rect>
```

5. Repeat steps 1-3 for the remaining **rect** elements. The final SVG markup code for all four **rect** elements should appear as per the following code.

```
<rect x="20" y="215" width="70" height="5" rx="2" ry="2" class="bar">
  <animate attributeName="y" fill="freeze" dur="3" begin="2" to="20" />
  <animate attributeName="height" fill="freeze" dur="3" begin="2" to="200" />
</rect>
<rect x="110" y="215" width="70" height="5" rx="2" ry="2" class="bar">
  <animate attributeName="y" fill="freeze" dur="3" begin="2" to="180" />
  <animate attributeName="height" fill="freeze" dur="3" begin="2" to="40" />
</rect>
<rect x="200" y="215" width="70" height="5" rx="2" ry="2" class="bar">
  <animate attributeName="y" fill="freeze" dur="3" begin="2" to="120" />
  <animate attributeName="height" fill="freeze" dur="3" begin="2" to="100" />
</rect>
<rect x="290" y="215" width="70" height="5" rx="2" ry="2" class="bar">
  <animate attributeName="y" fill="freeze" dur="3" begin="2" to="140" />
  <animate attributeName="height" fill="freeze" dur="3" begin="2" to="80" />
</rect>
```

6. Save the **Details.cshtml** file.

► Task 3: Add CSS styles to the SVG chart.

1. In Solution Explorer, expand the **Content\Styles** folder and open the **game.css** file.
2. Copy the content of the **svgStyles.txt** file found in the **D:\Mod06\Labfiles\Starter\Assets** folder to the end of the **game.css** file.



Note: The styles in the **svgStyles.txt** file control the look and layout of the chart. Take a quick look at the styles you just copied to the **game.css** file and note the use of styles such as **stroke** and **fill** that only apply to SVG documents.

3. Find the style rule for SVG text elements and add style properties to set the font and colors for *all* the **text** elements in the chart.

```
.chart-box text
{
  stroke: none;
  fill: #6D7EA9;
```

```
}    font-family: Arial;
```

4. Find the style rule matching the **bar** class and set the **stroke** and **stroke-width** style properties.

```
.chart-box .bar  
{  
    stroke: #000000;  
    stroke-width: 1;  
}
```

5. Save and close the **game.css** file.
6. Run the application by pressing F5 in Visual Studio.
7. Navigate to the **/Game/Details** page and observe the SVG bar chart with the SVG animation of the growing bars.

Results: Game statistics are integrated into the Game Details page.

Exercise 2: Adding Animation to the Game Details Canvas-Based Slideshow

► Task 1: Create an isolated time loop for the transition's animation.

1. In Solution Explorer, browse to the **Scripts** folder and open the **GameSlideshowScripts.js** file.
2. After the **slideshow** object, paste the content of the transition.txt file in the **D:\Mod06\Labfiles\Starter** folder to create the **transition** object.



Note: The **transition** object contains settings and data used to manage the animation.

1. At the end of the file, add the **startTransition** function and trigger the animation by calling **animateTransition**.

```
function startTransition() {
    animateTransition();
}
```

2. After the **startTransition** function, create the **animateTransition** function with the animation's isolated time loop.

```
function animateTransition() {
    window.setTimeout(animateTransition, transition.interval);
}
```

3. In the **animateTransition** function, before the **window.setTimeout** method call, increment the **transition** object's **currentStep** property every time the **animateTransition** function is called.

```
function animateTransition() {
    transition.currentStep = transition.currentStep + 1;
    window.setTimeout(animateTransition, transition.interval);
}
```

4. In the **animateTransition** function, only call the **window.setTimeout** method if the number of frames has not been exceeded.

```
function animateTransition() {
    transition.currentStep = transition.currentStep + 1;
    var steps = transition.steps;
    if (transition.currentStep < steps) {
        window.setTimeout(animateTransition, transition.interval);
    }
}
```

5. In the **rotateSlide** function, add a call to the **startTransition** function.

```
function rotateSlide() {
    // switch the selected slide to the next slide
    slideshow.currentSlideIndex = (slideshow.currentSlideIndex + 1) % slideshow.slideCount;
    startTransition();
}
```

► Task 2: Initialize the animation.

1. In the **startTransition** function, set the **transition.startLocation** property value to the current thumbnail's location.

```
function startTransition() {
  transition.startLocation = getThumbSlideLocation(slideshow.currentSlideIndex);
  animateTransition();
}
```

2. In the **startTransition** function, set the **transition.endLocation** property value to the main slide's location.

```
function startTransition() {
  transition.startLocation = getThumbSlideLocation(slideshow.currentSlideIndex);
  transition.endLocation = getMainSlideLocation();
  animateTransition();
}
```

3. In the **startTransition** function, set the **transition.currentStep** property value to 0.

```
function startTransition() {
  transition.startLocation = getThumbSlideLocation(slideshow.currentSlideIndex);
  transition.endLocation = getMainSlideLocation();
  transition.currentStep = 0;
  animateTransition();
}
```

► Task 3: Draw each frame of the animation.

1. In the **animateTransition** function, add a local variable to store the canvas's context.

```
if (transition.currentStep < steps) {
  var context = slideshow.context;
  window.setTimeout(animateTransition, transition.interval);
}
```

2. In the **animateTransition** function, create a path that contains the main slide area.

```
var context = slideshow.context;
context.beginPath();
context.rect(0, 0, slideshow.mainSlideWidth, slideshow.mainSlideHeight);
window.setTimeout(animateTransition, transition.interval);
```

3. In the **animateTransition** function, call the **context** object's **clip** method to ensure the transition is drawn only inside the main slide area.

```
var context = slideshow.context;
context.beginPath();
context.rect(0, 0, slideshow.mainSlideWidth, slideshow.mainSlideHeight);
context.clip();
window.setTimeout(animateTransition, transition.interval);
```

4. In the **animateTransition** function, paste the content of the **animateTransition.txt** file in the **D:\Mod06\Labfiles\Starter** folder. Paste the content after the call to the **context.clip** method. This will calculate the exact position to draw the current frame's image, relative to the **transition** object's **currentStep** property. The result should look like the following code.

```
context.clip();
var start = transition.startLocation;
var end = transition.endLocation;
var step = transition.currentStep;
var left = (start.left - end.left) / steps * (steps - step);
var top = (start.top - end.top) / steps * (steps - step);
```



```
var width = start.width + (end.width - start.width) / steps * step;
var height = start.height + (end.height - start.height) / steps * step;
window.setTimeout(animateTransition, transition.interval);
```

5. In the **animateTransition** function, get a reference to the current slide's image and draw it at the calculated position.

```
var height = start.height + (end.height - start.height) / steps * step;
var image = $(slideshow.context.canvas).find(".slide")[slideshow.currentSlideIndex].firstChild;
context.drawImage(image, left, top, width, height);
window.setTimeout(animateTransition, transition.interval);
```

► **Task 4: Prevent the main time loop from drawing, while the animation is in progress.**

1. Add the **canRefresh** property to the **slideshow** object and initialize it to **true**.



Note: Ensure that you add the comma at the end of the line with the interval: 5000 property setting; otherwise, errors may occur.

```
var slideshow = {
...
    interval: 5000,
    canRefresh: true
};
```

2. In the **startTransition** function, immediately before calling the **animateTransition** function, set the **slideshow** object's **canRefresh** property to **false**, to denote that an animation is in progress.

```
slideshow.canRefresh = false;
```

3. In the **animateTransition** function, immediately before the end of the function, add an **else if** block to set the **slideshow** object's **canRefresh** property to **true** when the transition ends, to denote that an animation is no longer in progress.

```
else if (transition.currentStep >= steps) {
    slideshow.canRefresh = true;
}
```

4. In the **drawMainSlide** function, surround everything, except for the call to the **window.setTimeout** method, with an **if** block that checks if the slideshow object's **canRefresh** property value is **true**. This ensures that the main slide is not redrawn while the transition is running, because that would "hide" the transition.

```
function drawMainSlide() {
    if (slideshow.canRefresh) {
        ...
    }
    // trigger the next iteration
    window.setTimeout(drawMainSlide, 20);
}
```

5. Save the **GameSlideshowScripts.js** file.
6. Run the application by pressing F5 in Visual Studio.

7. Navigate to the **/Game/Details** page and observe the new slideshow animation on the canvas.

Results: Animation is integrated into the Game Details page.

Module 7: Using Client-Side Storage

Lab: Using Client-Side Storage

Exercise 1: Creating the Cart Behavior by Using localStorage

► Task 1: Create the cart behavior.

1. Open Microsoft® Visual Studio® 2010.
2. Open the solution **GameStore.sln** from the **D:\Mod07\Labfiles\Starter** folder.
3. In case a Visual Studio dialog appears asking you whether to create a virtual directory click **Yes**.
4. In **Solution Explorer**, double-click the **GlobalScripts.js** file, which is located inside the **Scripts** folder in the **GameStore.MVC** project.
5. Add the following code to the end of the file.

```
var currentCart = null;
if (Modernizr.localStorage) {
}
```

6. In the **GlobalScripts.js** file, add the following **createEmptyCart** function at the end of the file.

```
function createEmptyCart() {
    localStorage.clear();
    localStorage.setItem("cart", JSON.stringify(new Cart()));
    currentCart = JSON.parse(localStorage.getItem("cart"));
}
```

7. In the code block of the **Modernizr localStorage** check, add the following code.

```
currentCart = JSON.parse(localStorage.getItem('cart'));
if (!currentCart) {
    createEmptyCart();
}
```

8. Press Ctrl+S to save the file.
9. Add the following code in the Modernizr code block after the last line of code that you wrote.

```
currentCart.getTotal = function () {
    var total = 0;
    // run on all the games, parse their price and add it to the total price the cart
    for (var i = 0; i < this.games.length; i++) {
        var strPrice = this.games[i].price.replace('$', '');
        var price = parseFloat(strPrice);
        total += price;
    }
    return total;
}
```

10. Press Ctrl+S to save the file.
11. Add the following code after the wire-up of the **getTotal** function.

```
// add an add game function to the current cart object
currentCart.addGame = function (game) {
```

```
// insert game into current cart's game array
currentCart.games.push(game);
// insert the new cart into the storage as string
localStorage.setItem('cart', JSON.stringify(currentCart));
}
```

12. Press Ctrl+S to save the file.

13. Add the following code after the wire-up of the **addGame** function.

```
// add an add game function to the current cart object
currentCart.removeGame = function (gameId) {
    var gameIndex;
    // search whether the gameId exists in the cart
    for (index in currentCart.games) {
        if (currentCart.games[index].gameId == gameId) {
            gameIndex = index;
            break;
        }
    }
    // if a gameId exists then use its index to remove it from the
    // cart html table and from the cart
    if (gameIndex) {
        // remove the row from the cart table
        document.getElementById("cartTable").deleteRow(gameIndex);
        // remove the game object from the current cart
        currentCart.games.splice(gameIndex, 1);
        // update the cart storage
        localStorage.setItem("cart", JSON.stringify(currentCart));
    }
}
```

14. Press Ctrl+S to save the file.

15. After the Modernizr code block, add the following **handleNoLocalStorage** function.

```
function handleNoLocalStorage() {
    $('#products').first().html('Your browser does not support local storage');
}
```

16. Press Ctrl+S to save the file.

17. After the **handleNoLocalStorage** function, add the **updateTotalPrice** function.

```
function updateTotalPrice() {
    $('#totalPrice').text('$' + currentCart.getTotal());
}
```

18. Press Ctrl+S to save the file.

Results: A JavaScript file containing all the relevant functions is created.

Exercise 2: Integrating the Cart with the Web Application

► Task 1: Integrate cart behavior on the Cart webpage.

1. In **Solution Explorer**, double-click the **CartScripts.js** file located inside the **Scripts** folder in the **GameStore.MVC** project.
2. Add the following **fillCartTable** function after the **contentLoaded** function.

```
function fillCartTable() {
    var table = document.getElementById("cartTable");
    for (var i = 0; i < currentCart.games.length; i++) {
        createTableRow(table, currentCart.games[i]);
    }
    updateTotalPrice();
}
```

3. After the **fillCartTable** function, add the following **createTableRow** function.

```
function createTableRow(table, game) {
    // create the new row
    var newRow = table.insertRow(table.rows.length);

    // create the game image cell
    var cell1 = newRow.insertCell(0);
    $(cell1).append($("<img width='48' height='65' />").attr('src',
game.imageUrl).attr('width', 63));

    // create the game title cell
    var cell2 = newRow.insertCell(1);
    $(cell2).attr('class', 'text').text(game.name).attr('width', 342);
    // create the link to the game details
    var cell3 = newRow.insertCell(2);
    var link = $('<a class="cart-detail-link" href="/ContosoGames/Game/Details/' + game.gameId
+ "'>Details</a>');
    $(cell3).append(link).attr('width', 325);
    // create the game price cell
    var cell4 = newRow.insertCell(3);
    $(cell4).attr('class', 'text').text(game.price).attr('width', 128);
    // create the remove from cart image
    var cell5 = newRow.insertCell(4);
    $(cell5).append("<img width='19' height='19' src='/ContosoGames/Content/Images/x.jpg'
/>").attr('data-game-id', game.gameId).attr('width', 88).click(function () {
        currentCart.removeGame(game.gameId);
        updateTotalPrice();
    });
}
```

4. Inside the **contentLoaded** function, add the following code.

```
if (Modernizr.localstorage) {
    fillCartTable();
}
else {
    handleNoLocalStorage();
}
```

5. Press Ctrl+S to save the file.

► **Task 2: Integrate the cart behavior on the Checkout webpage.**

1. In **Solution Explorer**, double-click the **CheckoutScripts.js** file located inside the **Scripts** folder in the **GameStore.MVC** project.
2. Add the following **fillCartTable** function after the **contentLoaded** function:

```
function fillCartTable() {
    var table = document.getElementById("cartTable");

    for (var i = 0; i < currentCart.games.length; i++) {
        createTableRow(table, currentCart.games[i]);
    }
    updateTotalPrice();
}
```

3. After the **fillCartTable** function, add the following **createTableRow** function:

```
function createTableRow(table, game) {
    // create the new row
    var newRow = table.insertRow(table.rows.length);
    // create the game image cell
    var cell1 = newRow.insertCell(0);
    $(cell1).append("<img width='48' height='65'/>").attr('src', game.imageUrl).attr('width', 117));
    // create the game title cell
    var cell2 = newRow.insertCell(1);
    $(cell2).attr('class', 'text').text(game.name).attr('width', 112);
    // create the game title cell
    var cell3 = newRow.insertCell(2);
    $(cell3).attr('class', 'text').text(game.price).attr('width', 71);
    // create the remove from cart image
    var cell4 = newRow.insertCell(3);
    $(cell4).append("<img width='19' height='19' src='/ContosoGames/Content/Images/x.jpg' />").attr('data-gameId', game.gameId).attr('width', 50).click(function () {
        currentCart.removeGame(game.gameId);
        updateTotalPrice();
    });
}
```

4. In the **contentLoaded** function, add the following code after the line with the **addPolyfills** function call.

```
if (Modernizr.localstorage) {
    fillCartTable();
}
else {
    handleNoLocalStorage();
}
```

5. Press Ctrl+S to save the file.

► **Task 3: Integrate the cart behavior on the GameDetails webpage.**

1. In **Solution Explorer**, double-click the **GameReviewScripts.js** file located inside the **Scripts** folder in the **GameStore.MVC** project.
2. Add the following **addAddToCartEventHandler** function at the end of the **GameReviewScripts.js** file.

```
function addAddToCartEventHandler() {
    $('add-to-cart-button').click(function () {
        var game = createGame();
        currentCart.addGame(game);
    });
}
```

3. In the **contentLoaded** function, add the call to **addAddToCartEventHandler** at the end of the function.

```
addAddToCartEventHandler();
```

4. Press Ctrl+S to save the file.

► **Task 4: Integrate the cart behavior on the Home webpage.**

1. In **Solution Explorer**, double-click the **HomepageScripts.js** file located inside the **Scripts** folder in the **GameStore.MVC** project.
2. Add the following **addAddToCartEventHandlers** function at the end of the **HomepageScripts.js** file.

```
function addAddToCartEventHandlers() {
    $('add-to-cart-button').each(function () {
        // add click event handler to all the add to cart buttons
        $(this).click(function () {
            var gameElement = $(this).parent()[0];
            insertRowToCartTable(gameElement);
        });
    });
}
```

3. Add the following **insertRowToCartTable** function after the **addAddToCartEventHandlers** function.

```
function insertRowToCartTable(gameElement) {
    // extract all the details for creation of a game object
    var gameImageUrl = gameElement.getElementsByTagName("img")[0].src;
    var gameName = gameElement.getElementsByClassName("game-text")[0].textContent;
    var gameId = gameElement.getAttribute("data-game-id");
    var price = gameElement.getElementsByClassName('money-text')[0].textContent;

    // create the game object
    var game = new Game(gameId, gameName, gameImageUrl, price);
    var table = document.getElementById("cartTable");
    createTableRow(table, game);
    currentCart.addGame(game);
}
```

4. In the **getAllGames** function, after the call to **collapseAllPanels** function, add the following code.

```
if (Modernizr.localstorage) {
    addAddToCartEventHandlers();
}
```

5. Press Ctrl+S to save the file.
6. In **HomepageScripts.js**, add the following **createEmptyCart** function.


```
function createEmptyCart() {
    localStorage.clear();
    localStorage.setItem("cart", JSON.stringify(new Cart()));
    currentCart = JSON.parse(localStorage.getItem("cart"));
}
```

7. After the **createEmptyCart** function, add the following **createCartTable** function.

```
function createCartTable() {
    var table = document.getElementById("cartTable");

    for (var i = 0; i < currentCart.games.length; i++) {
        createTableRow(table, currentCart.games[i]);
    }
}
```

8. After the **createCartTable** function, add the following **createTableRow** function.

```
function createTableRow(table, game) {
    // create the new row
    var newRow = table.insertRow(table.rows.length);

    // create the game image cell
    var cell1 = newRow.insertCell(0);
    $(cell1).append("<img width='48' height='65' />").attr('src', game.imageUrl));

    // create the game title cell
    var cell2 = newRow.insertCell(1);
    $(cell2).attr('class', 'text').text(game.name);

    // create the remove from cart image
    var cell3 = newRow.insertCell(2);
    $(cell3).append("<img width='19' height='19' src='/ContosoGames/Content/Images/x.jpg' />").attr('data-game-id', game.gameId).click(function () {
        currentCart.removeGame(game.gameId);
    });
}
```

9. In the **contentLoaded** function, add the following code before the function's closing brace {}.

```
if (Modernizr.localstorage) {
    createCartTable();
}
else {
    handleNoLocalStorage();
}
```

10. Press Ctrl+S to save the file.
11. Press Ctrl+F5 to run the application.



Note: Now, you can click the **Add To Cart** buttons to add games to the cart, dynamically. You can also remove games from the cart by clicking the X buttons in the cart. If you close the browser and run the application again, the cart with all its content (which is saved in the **localStorage**) will be available.

Results: The “Add to Cart” buttons on the Home, Game Details, and Cart pages include the cart implementation.

Module 8: Using Advanced HTML5 JavaScript APIs

Lab: Using Advanced HTML5 JavaScript APIs

Exercise 1: Integrating Drag-and-Drop Behavior into the Home Page

► Task 1: Integrate Drag-and-Drop Behavior.

1. Open Microsoft® Visual Studio® 2010.
2. Open the **GameStore.sln** solution from the **D:\Mod08\Labfiles\Starter** folder.
3. In case a message box appears prompting you to create a virtual directory for the project click **Yes**.
4. In **Solution Explorer**, expand the **GameStore.Mvc** project and the **Scripts** folder, and then double-click the **HomepageScripts.js** file.
5. Add the definition of the **addDragAndDropEventHandlers** function to the end of the file.

```
function addDragAndDropEventHandlers() {
}
```

6. Inside the **addDragAndDropEventHandlers** function, retrieve all the games by using their class name.

```
var games = document.getElementsByClassName("game");
```

7. In the **addDragAndDropEventHandlers** function, for every retrieved game element, wire the **dragstart** event.

```
// write the start drag event for all the games
for (var i = 0; i < games.length; i++) {
    games[i].addEventListener("dragstart", function (ev) {
        ev.dataTransfer.effectAllowed = 'copy';
        ev.dataTransfer.setData("text", this.getAttribute("data-game-id"));
    }, false);
}
```

8. In the **addDragAndDropEventHandlers** function, retrieve the **dropzone** element by using a **querySelector**.

```
// get the dropzone element
var dropElement = document.querySelector("#cart[dropzone]");
```

9. In the **addDragAndDropEventHandlers** function, wire the **dragover** event handler.

```
dropElement.addEventListener("dragover", function (ev) {
    if (ev.preventDefault) ev.preventDefault(); // allows us to drop
    ev.dataTransfer.dropEffect = "copy";
    return false;
}, false);
```

10. In the **addDragAndDropEventHandlers** function, wire the drop event handler.

```
dropElement.addEventListener("drop", function (ev) {
    if (ev.stopPropagation) ev.stopPropagation();
    var gameId = ev.dataTransfer.getData("text");
    var gameElement = document.querySelector("[data-game-id='" + gameId + "']");
```

```
insertRowToCartTable(gameElement);
    ev.stopPropagation();
    return false; // return false so the event will not be propagated to the browser
}, false);
```

11. In the **getAllGames** function, add a call to the **addDragAndDropEventHandlers** function after the call for **createCatalog** function and before the call for **addCollapsiblePanels** function. Wrap the call for **addDragAndDropEventHandlers** with a **Modernizr** check whether Drag-and-Drop feature is available in the browser:

```
if (Modernizr.draganddrop) {
    addDragAndDropEventHandlers();
}
```

12. Press Ctrl+S to save the file.
13. In the **HomepageScripts.js** file, in the **createCatalog** function, locate the creation of the game element (the statement that starts with `var gameElement`). Inside the article element string, add the **draggable** attribute so that the statement looks like the following code.

```
var gameElement = $('<article data-game-id="' + genres.genres[i].games[j].GameId + '"
class="game" draggable="true"><figure></figure><header><h3 class="game-
text">' + genres.genres[i].games[j].Name + '</h3></header><p class="money-text">' +
genres.genres[i].games[j].Price + '$</p><div class="button add-to-cart-button">Add to
Cart</div><a class="learn-more" href="/ContosoGames/Game/Details/' +
genres.genres[i].games[j].GameId + '">Learn More </a></article>');
```

14. Press Ctrl+S to save the file.
15. In **Solution Explorer**, expand the **GameStore.Mvc** project and the **Views\Home** folder, and then double-click the **Index.cshtml** file.
16. Locate the **aside** element with the **cart** id and add to it the **dropzone** attribute. The element starting tag should look like the following code.

```
<aside id="cart" class="box narrow" dropzone="true">
```

17. Press Ctrl+S to save the file.
18. Run the application by pressing Ctrl+F5. You should now be able to drag games to the cart.



Note: You must run the webpage in Internet Explorer 10 in order for Drag-and-Drop functionality to work.

Results: A Home Page with drag-and-drop behavior that enables the user to drag games from the Games catalog and drop them into the shopping cart.

Exercise 2: Add Geolocation Information to the Purchase Event

► Task 1: Create the Purchase Event.

1. In **Solution Explorer**, expand the **GameStore.Mvc** project and the **Scripts** folder, and then double-click **CheckoutScripts.js**.
2. At the end of the file, add the **addPurchaseEvent** function definition.

```
function addPurchaseEvent() {
}
```

3. In the **addPurchaseEvent** function, wire the **click** event by using **jQuery**.

```
$('#btnPurchase').click(function () {
});
```

4. At the end of the **contentLoaded** function, add a call for the **addPurchaseEvent** function.

```
addPurchaseEvent();
```

5. Press Ctrl+S to save the file.
6. In **Solution Explorer**, expand the **GameStore.Mvc** project and the **Scripts** folder, and then double-click the **GlobalScripts.js** file.
7. Add the definition of the **Location** object to the file.

```
var Location = (function (lat, long) {
    this.latitude = lat;
    this.longitude = long;
})();
```

8. Press Ctrl+S to save the file.
9. At the start of the **CheckoutScripts.js** file, add the following variable.

```
var buyerLocation = null;
```

10. Press Ctrl+S to save the file.
11. In the **CheckoutScripts.js** file, in the **click** event of the **Purchase** button (inside the **addPurchaseEvent** function), add the following code.

```
if (Modernizr.geolocation) {
    // get the current position and use it in the receipt
    navigator.geolocation.getCurrentPosition(function (e) {
        buyerLocation = new Location(e.coords.latitude, e.coords.longitude);
    });
}
```

12. In the **CheckoutScripts.js** file, add the following **makePurchase** function.

```
function makePurchase() {
    // Send a JSON GET request with the buyer location to the server side
    $.getJSON("/ContosoGames/Checkout/Purchase", { location: buyerLocation }, function
(response) {
        alert(response);
    });
}
```

```
});  
}
```

13. Call the **makePurchase** function after the setting of the **buyerLocation** object inside the **Geolocation** success callback. The **addPurchaseEvent** should look like the following code.

```
function addPurchaseEvent() {  
    $('#btnPurchase').click(function () {  
        if (Modernizr.geolocation) {  
            // get the current position and use it in the receipt  
            navigator.geolocation.getCurrentPosition(function (e) {  
                buyerLocation = new Location(e.coords.latitude, e.coords.longitude);  
                makePurchase();  
            });  
        }  
    });  
}
```

14. In the **Index.cshtml** file, locate the **div** element with the **class** attribute whose value is "button". Add an **id** attribute and set its value to "btnPurchase".

```
<div class="button" id="btnPurchase">  
<a href="#">Purchase</a>  
</div>
```

15. Press Ctrl+S to save the file.
16. Run the application by pressing Ctrl+F5. Go to the checkout form and click the **Purchase** button. An alert will appear if the request to the server succeeds.

Results: The purchase data that is sent to the server contains the location data.

Exercise 3: Add Coupon-Reading Behavior

► Task 1: Add a file input type to the Checkout webpage.

1. In **Solution Explorer**, expand the **GameStore.Mvc** project and the **Views\Checkout** folder, and then double-click the **Index.cshtml** file.
2. In the Credit Card Information **fieldset**, before the closing tag of the **div** element, add the following definition for a **file** input type.

```
<p>
<label class="field-name" for="CouponField">Add Coupon</label>
<input name="Coupon" type="file" class="field-value" id="CouponField" />
</p>
```

3. Press Ctrl+S to save the file.

► Task 2: Add the Coupon-Reading Functionality.

1. In **Solution Explorer**, expand the **GameStore.Mvc** project and the **Scripts** folder, and then double-click **CheckoutScripts.js**.
2. In the **CheckoutScripts.js** file, add the following **readCoupon** function.

```
function readCoupon() {
    if (window.File && window.FileReader) {
        var files = document.getElementById('CouponField').files;
        if (files.length > 0) {
            // create a new FileReader object to handle the reading of the file
            var reader = new FileReader();
            // create a handler to output the content of the file to the user
            reader.onload = function (e) {
                // show the coupon content inside an alert
                alert(e.target.result);
            };
            // read in the text file as a text
            reader.readAsText(files[0]);
        }
    }
}
```

3. In the **addPurchaseEvent** function, add a call to **readCoupon** inside the **Geolocation** success callback. The function should look like the following code.

```
function addPurchaseEvent() {
    $('#btnPurchase').click(function () {
        if (Modernizr.geolocation) {
            // get the current position and use it in the receipt
            navigator.geolocation.getCurrentPosition(function (e) {
                buyerLocation = new Location(e.coords.latitude, e.coords.longitude);
                readCoupon();
                makePurchase();
            });
        }
    });
}
```

4. Press Ctrl+S to save the file.
5. In **Solution Explorer**, expand the **GameStore.Mvc** project and the **Views\Checkout** folder, and then double-click the **Index.cshtml** file.

6. Run the application by pressing Ctrl+F5. Go to the Checkout page. Click the **Browse** button to select a local text file. Click **Purchase**, and you should see the contents of the text file in an alert.

Result: When the **Purchase** button is clicked, an alert with the text data in the coupon text file is shown on screen.

Module 9: Using WebMatrix and Other Developer Tools

Lab: Using WebMatrix and Other Development Tools

Exercise 1: Using WebMatrix to Create a Website

► **Task 1: Create a new website.**

1. On the Windows Start menu, click **All Programs**, click the **Microsoft WebMatrix** folder, and then click the **Microsoft WebMatrix** program icon.
2. On the WebMatrix **Quick Start** screen, click **Site From Template**.
3. Select the **Empty Site** template, name it **WebMatrixQuickStart**, and then click **OK**.
4. In the Workspace selection pane, click **Files** to switch to the Files workspace.

► **Task 2: Add existing items to the project.**

1. To create a new project folder, in the Navigation pane, right-click the **WebMatrixQuickStart** folder, and then click **New Folder**.
2. Type **Images**, and then press **Enter**.
3. On the **Home** tab, in the **Files** group, click **Add Existing**.
4. In the **Add Existing File** dialog box, navigate to **D:\Mod09\Labfiles\Assets\WebMatrix\Images**, select all the images in the folder, and then click **Open**.
5. To create a new project folder, in the Navigation pane, right-click the **Images** folder, and then click **New Folder**.
6. Type **Games**, and then press **Enter**.
7. On the **Home** tab, in the **Files** group, click **Add Existing**.
8. In the **Add Existing File** dialog box, navigate to **D:\Mod09\Labfiles\Assets\WebMatrix\Images\Games**, select all the images in the folder, and then click **Open**.
9. To create a new project folder, in the Navigation pane, right-click the **WebMatrixQuickStart** folder and click **New Folder**.
10. Type **Scripts** and press **Enter**.
11. On the **Home** tab, in the **Files** group, click **Add Existing**.
12. In the **Add Existing File** dialog box, navigate to **D:\Mod09\Labfiles\Assets\WebMatrix\Scripts**, select all the JavaScript files in the folder, and then click **Open**.
13. In the Navigation pane, right-click the **WebMatrixQuickStart** folder and select the **New Folder** command to create a new project folder.
14. Type **Styles**, and then press **Enter**.
15. On the **Home** tab, on the ribbon, in the **Files** group, click **Add Existing**.

16. In the **Add Existing File** dialog box, navigate to **D:\Mod09\Labfiles\Assets\WebMatrix\Styles**, select all the CSS files in the folder, and then click **Open**.
17. In the Navigation pane, select the **WebMatrixQuickStart** folder, and then, on the **Home** tab, in the **Files** group, click **Add Existing**.
18. In the **Add Existing File** dialog box, select the **D:\Mod09\Labfiles\Assets\WebMatrix\web.config** file and then click **Open**.

► **Task 3: Add a new HTML file to the project and edit its content.**

1. In the Navigation pane, select the **WebMatrixQuickStart** folder, and then, on the **Home** tab, in the **Files** group, click the **New** icon (not the button with the arrow).
2. In the **Choose a File Type** dialog box, select the **HTML** file, name it **Homepage.html**, and then click **OK**.
3. In the **title** element, add the text, **Contoso, Ltd**.

```
<title>Contoso, Ltd</title>
```

4. In the **head** element, after the **title** element, add **link** elements to reference the CSS files and **script** elements to reference the JavaScript files.

```
<link href="Styles/theme.css" rel="stylesheet" type="text/css" />
<link href="Styles/slideshow.css" rel="stylesheet" type="text/css" />
<link href="Styles/spotlight.css" rel="stylesheet" type="text/css" />
<link href="Styles/catalog.css" rel="stylesheet" type="text/css" />
<link href="Styles/cart.css" rel="stylesheet" type="text/css" />
<link href="Styles/dragdrop.css" rel="stylesheet" type="text/css" />
<scriptsrc="Scripts/jquery-1.5.1.js" type="text/javascript"></script>
<scriptsrc="Scripts/GlobalFunctions.js" type="text/javascript"></script>
```

5. Log on to the **SEA-DEV** virtual machine as **Admin**, with the password, **Pa\$\$w0rd**.
6. Open the **body markup.txt** file from the **D:\Mod09\Labfiles\Assets\WebMatrix** folder.
7. Copy the **body markup.txt** file contents to the clipboard by selecting it all and pressing **Ctrl+C**.
8. Paste the HTML code into **Homepage.html** between the opening and closing **<body>** tags by placing the cursor on the line between the tags and pressing **Ctrl+V**.

► **Task 4: Run the website.**

1. To save all the modified files, on the Quick Access toolbar, click the **Save All** button.
2. In the Navigation pane, select the **Homepage.html** file.
3. To preview the POC site in the default web browser, on the Home tab, in the **Site** group, click **Run**.

Results: A WebMatrix POC project is created with the layout of the first module's lab solution.

Exercise 2: Using ASP.NET MVC 3 to Create a Website

► Task 1: Create and configure a new ASP.NET MVC 3 project.

1. Start Visual Studio 2010.
2. On the **File** menu, click **New**, and then click **Project**.
3. In the **Installed Templates** folder tree, expand **Visual C#\Web** and select the **ASP.NET MVC 3 Web Application** project template.
4. In the **Name** field, type **MVCQuickStart**.
5. Click **OK**.
6. In the **New ASP.NET MVC 3 Project** dialog box, select the **Empty** template and then select the **Razor** view engine if it is not already selected.
7. Click **OK**.
8. In Solution Explorer, right-click the project name **MVCQuickStart** and click **Properties**.
9. Click the **Web** tab.
10. Select **Use Local IIS Web Server**, if not already selected.
11. Check the **Use IIS Express** check box, if not already checked.
12. Add **ContosoGames** to the end of the **Project Url**.
13. Click **Create Virtual Directory**.
14. In Visual Studio, a confirmation dialog box appears indicating the successful creation of the virtual directory.
15. To close the dialog box, click **OK**.
16. Save and close the project properties.

► Task 2: Add existing assets and modify the CSS and CSHTML files.

1. In Solution Explorer, under the **MVCQuickStart** project node, right-click the **Scripts** folder, click **Add**, and then click **Existing Item**.
2. Navigate to **D:\Mod09\Labfiles\Assets\MVC\Scripts**, select the **GlobalFunctions.js** file, and then click **Add**.
3. Use Windows Explorer to locate **D:\Mod09\Labfiles\Assets\MVC\Images**, drag the **Images** folder to Solution Explorer in Visual Studio, and drop the **Images** folder on the **Content** folder.



Note: Alternatively, you can right-click the **Images** folder in Windows Explorer, click **Copy** and then paste it into the **Content** folder in Solution Explorer in Visual Studio.

4. Save the project by pressing **Ctrl+Shift+S**.
5. In Solution Explorer, in the **Content** folder, select the **Site.css** file.
6. Press **Del** to delete the **Site.css** file, and then click **OK** in the confirmation dialog box.
7. In Solution Explorer, in the **Content** folder, select the **themes** folder.

8. Press **Del** to delete the **themes** folder, and then, in the confirmation dialog box, click **OK**.
9. Use Windows Explorer to locate the **D:\Mod09\Labfiles\Assets\MVC\Styles** folder, drag the **Styles** folder to Solution Explorer in Visual Studio, and then drop the **Styles** folder on the **Content** folder.
10. Save the project by pressing **Ctrl+Shift+S**.
11. In Solution Explorer, locate and open the **_Layout.cshtml** template file in the **Views\Shared** folder.
12. In the **head** element, replace the **link** element with a **link** element by referencing the **theme.css** file.

```
<link href="@Url.Content("~/Content/Styles/theme.css")" rel="stylesheet" type="text/css" />
```

13. Add a **Styles** section where pages using the template can add additional CSS files, after the **link** element.

```
@RenderSection("Styles", false)
```

14. Add a reference to the **GlobalFunctions.js** file before the closing **</head>** tag.

```
<script src="@Url.Content("~/Scripts/GlobalFunctions.js")" type="text/javascript"></script>
```

15. Save **_Layout.cshtml** by pressing **Ctrl+Shift+S**.

► Task 3: Create a model.

1. In Solution Explorer, right-click the **Models** folder, click **Add**, and then click **New Item** from the context menu.
2. In the **Add New Item** dialog box, in the **Installed Templates** tree, click **Visual C#**, select the **Class** template, name it **ContosoGamesModel.cs**, and then click **Add**.
3. Replace the auto-generated class code with the following **Game** class definition.

```
public class Game
{
    public int GameId { get; set; }
    public string Name { get; set; }
    public string Genre { get; set; }
    public string ShortDescription { get; set; }
    public decimal Price { get; set; }
    public string GameImageUrl { get; set; }
}
```

4. Add the following class definition to the **ContosoGamesModel.cs** file, below the **Game** class.

```
public class GameStoreContext
{
    public static List<Game> Games { get; private set; }
    static GameStoreContext()
    {
        InitGameList();
    }
    public static void InitGameList()
    {
        Games = new List<Game>
        {
            new Game
            {
                GameImageUrl = "/ContosoGames/Content/Images/Games/BlueDragon_Box.png",
                Genre = "Role-Playing",
            }
        }
    }
}
```

```

        Name = "Blue Dragon",
        Price = 18,
        ShortDescription = "Blue Dragon is an unexceptional RPG that does just enough
to escape mediocrity.",
    },
    new Game
    {
        GameImageUrl = "/ContosoGames/Content/Images/Games/Forza2Box.jpg",
        Genre = "Sports",
        Name = "Forza MotorSport 2",
        Price = 20,
        ShortDescription = "The definitive racing game.",
    },
    new Game
    {
        GameImageUrl = "/ContosoGames/Content/Images/Games/Kameo_Box.jpg",
        Genre = "Action and Adventure",
        Name = "Kameo: Elements of Power",
        Price = 20,
        ShortDescription = "Unleash the powers within you!",
    },
    new Game
    {
        GameImageUrl = "/ContosoGames/Content/Images/Games/Pes6_Box.jpg",
        Genre = "Sports",
        Name = "PES 2011",
        Price = 30,
        ShortDescription = "Pro Evolution Soccer has always been regarded as the
quality soccer title.",
    }
    };
}
}

```

5. Save the **ContosoGamesModel.cs** file by pressing **Ctrl+S**, and then close the file.

► Task 4: Create a controller.

1. In Solution Explorer, right-click the **Controllers** folder, click **Add**, and then click **Controller**.
2. In the **Add Controller** dialog box, change the controller name to **HomeController**. Then, from the **Template** drop-down box, select **Empty Controller**, and then click **Add**.
3. Add the following **using** statement to the end of the file's using statements block.

```
using MVCQuickStart.Models;
```

4. Replace the **Index** action method body with the following code.

```

// get all the games from the game store context
var games = GameStoreContext.Games;
// create a model for the view that will contain all the games divided by
// their genres
var genres = new Dictionary<string, List<Game>>();
// insert all the games into their relevant genre
foreach (var game in games)
{
    if (genres.ContainsKey(game.Genre))
    {
        genres[game.Genre].Add(game);
    }
    else
    {

```

```
        genres.Add(game.Genre, new List<Game> { game });  
    }  
    }  
    return View(genres);  
}
```

5. Save the file by pressing **Ctrl+S**, and then close the file.

► **Task 5: Create a view.**

1. In Solution Explorer, right-click the **Views** folder, click **Add**, and then click **New Folder**.
2. To set the new folder name, type **Home**, and then press **Enter**.
3. Right-click the new **Home** folder, click **Add**, and then click **View**.
4. In the **Add View** dialog box, change the view name to **Index**, select the **Razor(CSHTML)** view engine, and then click **Add**.
5. Use any text editor (such as Notepad) to open the **View Markup.txt** file located in the **D:\Mod09\Labfiles\Assets\MVC** folder.
6. Replace all the contents in the **Index.cshtml** view file with the contents of the **View Markup.txt** file.
7. Examine the code. See how **@model** defines the view's model, how the **Styles** section is generated, and how the **catalog** section iterates over genres in the model, and then over games in each genre, all the while using Razor syntax to generate HTML elements for each genre and for each game.
8. Save the file by pressing **Ctrl+S**.

► **Task 6: Run the website.**

1. Compile and run the project by pressing **F5**.
2. Check that the created site works as expected by expanding and collapsing the genres. Notice that the games in each genre are generated by using Razor.

Results: An ASP.NET MVC 3 POC project is created with the layout of the first module's lab solution.